UNIVERSITÉ DU QUÉBEC À MONTRÉAL

UN LANGAGE SPÉCIFIQUE AU DOMAINE POUR L'OUTIL DE CORRECTION DE TRAVAUX DE PROGRAMMATION OTO

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

PAUL LESSARD

FÉVRIER 2010

REMERCIEMENTS

Avant tout, et plus que quiconque, j'aimerais remercier mon directeur de recherche, le professeur Guy Tremblay, pour sa confiance, son soutien et pour avoir cru en moi au cours des dernières années, malgré les tâches supplémentaires associées à son mandat de directeur du Département d'informatique de l'UQAM. Je le remercie également pour son soutien financier, qui m'a permis entre autres de bénéficier de bourses FARE. Aussi, un merci tout particulier à Frédéric Guérin, l'auteur principal de la première version d'Oto, même s'il a choisi d'embrasser un domaine autre que l'informatique. Si jamais tu lisais ces lignes, Frédéric, saches que je suis très admiratif de ton travail avec Oto et que c'est avec le plus grand respect que je me suis donné pour mission de le parfaire à la lumière de l'expérience acquise à l'utilisation de ta première version. Par ailleurs, je remercie aussi les professeurs chargés de corriger ce mémoire.

Ensuite, je me permettrai de saluer mes collègues de la maîtrise et du doctorat en informatique, entre autres Jean-Sébastien Gélinas (pour les dîners et autres moments innombrables où nous avons procrastiné ensemble), Abderrahmane Leshob (un des meilleurs équipiers avec qui j'ai eu la chance de collaborer), Dunarel Badescu (qui, pour moi, est l'incarnation de la passion de l'informatique) ainsi que Romdhane Ben Younes et Maryem Ourti. Mentionnons également Stéphanie Lanthier, du Sitel, pour nos discussions délirantes et son aide lorsque nous avons migré Oto vers plusieurs machines du Labunix alors en chantier.

Je tiens également à remercier ma famille pour son support et ses encouragements depuis toujours, notamment au cours de mes études collégiales, de mon baccalauréat et encore lors de ces dernières années de maîtrise. Sans vous tous, je n'y serais pas parvenu. Merci à ceux d'entre vous qui ont lu mon mémoire, et ainsi qu'à ceux qui, pour me faire plaisir, ont tenté de le lire.

Finally, a very special thank you will go to my beloved fiancée, Catherine Wilkins. Catherine, I love you to pieces, and to you I dedicate this master's thesis on which you have seen me working so often lately. By your side, I feel like I could move mountains. Thank you for your infinite tolerance, your patience, your guidance and your lovely presence during this endeavour. After annoying you an insane amount of times with Oto, let's indulge ourselves with some well-deserved time together before considering whether or not to go for more! ♡

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

L'utilisation d'un outil d'aide à la correction peut contribuer à faciliter et à accélérer la correction de travaux pratiques de programmation. Au fil des ans, plusieurs logiciels ont été développés à cette fin, mais souffraient souvent d'un manque de flexibilité et étaient limités à un seul langage cible. Face à cette situation, l'outil Oto, développé à l'UQAM en langage Ruby, se voulait générique et extensible, pouvant théoriquement appliquer n'importe quel test à n'importe quel travail pratique. Toutefois, l'utilisation d'Oto en situation réelle a permis de constater que l'outil souffrait de certains défauts susceptibles de nuire à son utilisation à grande échelle, notamment en raison du manque de flexibilité de ses scripts de correction et de ses performances relativement faibles.

Dans ce mémoire, nous présentons les modifications que nous avons apportées à Oto pour en améliorer la flexibilité et les performances. D'abord, nous avons analysé l'outil pour en comprendre le fonctionnement et en localiser les faiblesses, qui provenaient principalement de choix architecturaux de coordination des tâches de correction. Nous nous sommes ensuite attardés sur les caractéristiques et les capacités qui devraient être offertes par Oto. Notre solution est passée par le remplacement du mécanisme de scripts Oto par un langage spécifique au domaine de type interne (*internal DSL*) basé sur Ruby. Au meilleur de nos connaissances, nous sommes les premiers à avoir eu recours à une telle approche dans le cadre d'un outil d'aide à la correction.

Les résultats que nous avons obtenus avec le DSL Oto sont venus confirmer une hausse considérable de la flexibilité et une amélioration des performances de l'outil, particulièrement pour les scripts de correction ayant recours à un nombre élevé de courts tests et aux corrections intra-groupes.

INTRODUCTION

Quiconque en a déjà fait l'expérience sait que corriger des travaux d'étudiants n'est pas une tâche triviale. Lorsqu'il s'agit de travaux pratiques de programmation, sa difficulté se voit encore augmentée, car en plus de vérifier des textes imprimés, le correcteur devra souvent manipuler des fichiers remis sur un support électronique et en vérifier le fonctionnement dynamique. Face à cette situation, l'intérêt d'un logiciel d'aide à la correction des travaux de programmation, permettant d'automatiser une partie de cette tâche, apparaît clairement. Bien que plusieurs applications destinées à cette fin ont été développées au fil des ans, elles étaient habituellement limitées dans leurs capacités de correction. À l'opposé, Oto (17; 33; 35) se distingue par une conception ayant pour buts premiers la généricité et l'extensibilité.

Développé à l'UQAM dans le cadre de la maîtrise en informatique de Frédéric Guérin sous la direction du professeur Guy Tremblay, Oto apportait des possibilités nouvelles aux outils de correction, permettant, en théorie, de corriger tout travail pratique en lui imposant n'importe quel test permis par un mécanisme de modules d'extension. Si le projet de maîtrise de Guérin a pu montrer la faisabilité d'un tel système (17), l'utilisation d'Oto dans le cadre de tests véritables et nos analyses de son code source ont démontré que la version initiale de l'outil souffrait de certains défauts, dont deux surtout étaient susceptibles de nuire à une utilisation de l'outil pour accomplir les tâches pour lesquelles il avait été conçu.

D'abord, Oto souffrait de problèmes de flexibilité. Reléguer le travail de correction à des modules d'extension souffrait des défauts de ses qualités : la nécessité de disposer d'un module pour effectuer toute tâche, alors que pour des opérations simples telles que de copier des fichiers, une commande du shell Unix (`cp`) aurait suffi.[1] De plus, la description des tâches, c'est à dire les instructions données à Oto pour lui indiquer comment manipuler ses modules et résultats, se faisant dans un langage maison (nommé OtoScript), il était difficile de convaincre des utilisateurs potentiels d'Oto de se lancer dans son apprentissage, car cela représentait pour eux un effort supplémentaire, qu'ils considéraient potentiellement superflu (puisque utilisable

---

[1] Si, par la suite, Oto a disposé d'un module permettant de lancer une commande shell quelconque, celui-ci était néanmoins d'une utilisation lourde et peu intuitive.

qu'avec ce seul outil). Ironiquement, OtoScript était un langage réduit, ne disposant pas de certaines structures couramment disponibles dans les langages de programmation, telles que les boucles. L'absence de telles structures complexifiait le développement et la compréhension des scripts et limitait leurs capacités, contraignant l'utilisateur à mêler ses appels à Oto à des scripts shell. Le plus grand problème était toutefois le modèle d'exécution de ces scripts et la manipulation des modules, où chacun des travaux était corrigé de manière isolée et individuelle. Une telle vision d'ensemble réduisait les possibilités de tâches de correction où les travaux étaient comparés entre eux.

Un autre problème limitant l'utilisation d'Oto se trouvait au niveau des performances de sa version initiale. Parfois, Oto pouvait prendre plusieurs minutes pour retourner le résultat d'une correction, ce qui était problématique tant pour un enseignant ayant un groupe entier à corriger[2] que pour un étudiant qui souhaitait utiliser Oto dans le cadre d'un laboratoire noté limité dans le temps, tels que ceux des premiers cours de programmation à l'UQAM.[3] Or, comme nous étions convaincus que l'utilisation d'Oto dans le cadre d'un laboratoire noté serait profitable pour les étudiants, tel que motivé par les travaux de Becker (6), régler sa lenteur devenait un objectif important.

De manière générale, nous pouvons affirmer que les défauts d'Oto nuisaient à son utilisation à une plus grande échelle et occultaient ses qualités intrinsèques prometteuses.[4] Satisfaire les objectifs de flexibilité et de performance dont nous souhaitions doter Oto devenait dès lors un nouveau défi de recherche intéressant au niveau de la maîtrise en informatique, tâche que le professeur Tremblay, notre directeur de recherche, nous a confiée.

Dans ce mémoire, nous discuterons des améliorations que nous avons apportées à Oto et des décisions et tâches qui nous ont menés à celles-ci. Dans le chapitre 1, nous discuterons d'outils d'aide à la correction et d'Oto de manière générale : sa place parmi les outils d'aide à la correction, ses concepts, les termes qu'il a introduits. Nous présenterons aussi certains des pro-

---

[2]Si le délai ne remettait pas en question la pertinence de la correction, il pouvait rendre pénible le débogage d'une tâche de correction lorsque plusieurs minutes étaient nécessaires pour exposer une erreur dans le script la décrivant.

[3]Étant donné que l'étudiant qui attendrait les résultats d'Oto perdra du temps, et celui qui ne les attendrait pas perdrait les bénéfices du *feedback* rapide lié à l'utilisation de l'outil.

[4]D'autant plus qu'elles étaient et demeurent, à notre connaissance, inégalées par ses pairs.

jets liés à Oto qui ont été développés après la mise au point de son noyau, notamment un module de détection du plagiat que nous avons développé au cours de notre apprentissage de l'outil. Au chapitre 2, nous analyserons la mise en œuvre et les performances des versions initiales d'Oto en fonction des objectifs que le professeur Tremblay nous avait fixés. Cette analyse se fera en deux temps. D'abord, nous discuterons de manière générale de la construction de l'application à partir de nos inspections manuelles et du portage d'Oto vers différentes machines de type Unix de l'UQAM. Ensuite, nous en identifierons les défauts qui seront traités lors des chapitres suivants. Notre solution passera par le développement d'un nouveau langage de description des tâches de correction basé sur Ruby, plus flexible et s'exécutant plus rapidement qu'OtoScript. Le chapitre 3 présentera de manière théorique l'approche que nous avons choisie pour le développement de notre nouveau langage, soit un langage spécifique au domaine de type interne (*internal DSL*). Signalons qu'afin d'éviter d'employer un néologisme qui nous serait propre, nous nous référerons aux « langages spécifiques au domaine » par l'abréviation DSL, de l'anglais *domain specific language*, qui est courante dans la littérature. Le chapitre 4 présentera la mise en œuvre de notre DSL ainsi que les modifications que nous avons apportées à Oto, tant au niveau de sa structure interne, de ses modules d'extension que de ses tests. Nous y traiterons également des problèmes auxquels nous avons fait face au cours de ces modifications et des solutions que nous avons apportées. Les deux derniers chapitres serviront à décrire les gains apportés par nos modifications par rapport aux objectifs de flexibilité et de performances. Le chapitre 5 servira essentiellement à donner des exemples des nouvelles capacités d'Oto. Finalement, au chapitre 6, nous aborderons la question des performances de l'outil et des accélérations apportées par nos changements.

CHAPITRE I

OTO, UN OUTIL POUR CORRIGER LES TRAVAUX DE PROGRAMMATION

Qu'est-ce qu'Oto? Quelles en sont les origines? Comment Oto permet-il de corriger des travaux? Quelles améliorations ont-elles été faites à Oto depuis ses débuts? Quelles applications ont-elles été associées à Oto pour faciliter son utilisation?

Dans ce premier chapitre, nous introduirons Oto. D'abord, nous considérerons la question des outils d'aide à la correction en accordant une attention particulière aux outils développés ces dernières années. Nous présenterons ensuite Oto de manière générale, en en décrivant les termes, les concepts et les capacités théoriques. Ensuite, nous présenterons le travail qui a été fait au fil des ans pour en étendre les capacités et faciliter son utilisation. Une attention particulière sera apportée à un ajout que nous avons fait au début de notre apprentissage de l'outil pour lui ajouter la capacité de détecter le plagiat intra-groupe.

Pour le reste de ce mémoire, nous nous référerons à Oto à l'aide du terme « version 1 » pour désigner la version d'origine décrite dans le mémoire de Frédéric Guérin (17). La version augmentée des modifications apportées par le professeur Tremblay, de l'interface Web développée par Mohamed Takim (31), du module de détection du plagiat et de certaines commandes sera identifiée par le terme « version 1+ ». Lorsque nous souhaiterons désigner ces deux premières versions à la fois, nous emploierons le terme « versions initiales ». La « version 2 » désignera notre propre version d'Oto, après les modifications majeures que nous lui avons apportées dans le cadre de notre travail de recherche.

1.1 Des outils d'aide à la correction

L'histoire des outils d'aide à la correction est presque aussi longue que celle de l'enseignement de la programmation. Dans son mémoire de maîtrise (17), Frédéric Guérin en a fourni un très bon résumé, que nous présenterons brièvement dans cette section.

D'abord, les premiers outils procédaient par inclusion, c'est à dire que le code de l'étudiant était intégré à celui de l'enseignant avant d'être compilé, lié et exécuté. Une routine de l'étudiant pouvait être ajoutée manuellement par l'enseignant à son propre code (12) ou pouvait appeler elle-même la routine de l'enseignant, déchargant la tâche de celui-ci (11). Par la suite, les approches utilisées furent basées sur la comparaison textuelle. Le programme de l'enseignant fournissait à celui de l'étudiant certaines entrées et comparait les sorties obtenues avec la bonne réponse. Ce fut apparemment le système BAGS (18) qui introduisit ce concept en 1969, lequel représentait un pas en avant par rapport à l'inclusion en découplant complètement le code de l'étudiant et de l'enseignant. La comparaison textuelle souffrait néanmoins d'un grave défaut : les étudiants devaient respecter à la lettre les instructions de leur enseignant au niveau du format des entrées et sorties attendues. Une solution était d'offrir aux étudiants la possibilité d'exécuter la version de l'enseignant. Ils pouvaient ensuite utiliser cette dernière comme modèle à reproduire. Certains systèmes, notamment Ceilith (12), ont eu recours a cette approche.

Les années 2000 ont vu l'apparition d'un nouveau procédé de correction, l'analyse réflexive. Celle-ci a permis d'étendre la comparaison à la vérification de valeurs de retour de méthodes Java dans le système BOSS2 (20). Celle-ci était avantageuse au niveau des valeurs pouvant être comparées, qui n'étaient plus limitées aux seules chaînes de caractères, mais pouvaient s'appliquer à n'importe quel objet Java. OCETJ, développé à l'UQAM dans le cadre de la maîtrise en informatique d'Éric Labonté sous la direction du professeur Guy Tremblay (22), visait à étendre cette approche en développant un outil basé sur le cadre de tests unitaires JUnit (5). Dans OCETJ, l'enseignant crée des exercices et associe à chacun un test public (avec lequel l'étudiant vérifie préalablement son travail) et un test privé (une vérification plus complète destinée à l'enseignant lui-même). Un autre système, Web-CAT (10), permettait aux étudiants de développer leur autonomie en créant eux-mêmes leurs tests JUnit et à vérifier leurs travaux avec ceux-ci. Guérin concluait sa revue de littérature par une mise en garde adressée par Stephen Edwards (10) à propos des tests fournis par l'enseignants, qui pouvaient entraîner les étudiants à adopter une approche par essais et erreurs au lieu de chercher véritablement à

comprendre la nature du problème à résoudre.

Au delà des différentes approches auxquelles ils avaient recours, les différents outils de correction ayant vu le jour au fil des ans souffraient de manque d'adaptabilité, de flexibilité et de généricité. La plupart d'entre eux étaient limités à un seul langage de programmation. BOSS2 et OCETJ, par exemple, s'appliquaient au seul langage Java et se limitaient aux tests de type réflexifs. Or, l'idéal aurait été de disposer d'un outil pouvant s'adapter à diverses situations de correction et ayant recours à différentes approches à cet effet. C'est l'approche utilisée par Oto (17; 33; 35), qui cherchait théoriquement à être adaptable à toute situation de correction, étant notamment programmable au moyen de scripts.

La question des outils d'aide à la correction a continué d'évoluer ces dernières années. La plupart des outils demeurent utilisés par un seul établissement, ou même un seul département informatique. OCETJ, par exemple, était limité au contexte du Cégep du Vieux-Montréal où les technologies sur lesquelles il était basé étaient utilisées. La nécessité pour un outil d'aide à la correction d'être générique et de pouvoir être adapté aux besoins divers d'institutions d'enseignements différentes fut l'une des conclusions de Douce, Livingstone et Orwell (9).

La question de la facilité de l'utilisation des outils d'aide à la correction par les étudiants a fait l'objet d'un article par Allowatt et Edwards en 2005 (2). Leur travail a consisté à intégrer l'outil Web-CAT à l'environnement de développement Eclipse pour les langages C++ et Java. Si leur approche s'est avérée un succès, elle exigeait toutefois l'installation de plusieurs logiciels sur le poste de travail utilisé, ce qui limitait l'utilisation de l'extension Eclipse sur les machines personnelles des étudiants. Une application peut également être utilisée indépendamment d'un environnement de développement intégré. C'est le cas de CourseMaker (19), venu remplacer Ceilith, qui se présente comme une application Java munie d'une interface *Swing*.

Au niveau de la définition des tâches de correction, l'approche utilisée par Oto a récemment (2008) été reprise par le système ProtoAPOGEE (14), dont les tests sont définis à l'aide de scripts Ruby Watir.[1] À l'instar de ses contemporains Oto et CourseMaker, cet outil permet aux étudiants d'obtenir du *feedback* rapide sur leur travail.

---

[1]Un outil d'automatisation de tests Web en Ruby. `http://wtr.rubyforge.org/`

1.2 Oto, un outil générique et extensible

Oto est un outil générique et extensible, développé en langage Ruby sous Unix, destiné à corriger les travaux de programmation. Ce sont ces deux qualités qui le distinguent par rapport aux autres outils qui ont vu le jour avant lui. Contrairement à ceux-ci, Oto ne se limite pas à une seule approche en terme de correction (comparaison textuelle des entrées/sorties par rapport à un modèle prédéfini, analyse introspective d'objets au moyen d'assertions, etc.) Au contraire, Oto peut éventuellement être utilisé pour chacune de ces approches, ce qui lui confère un potentiel intéressant.

De manière générale, Oto dispose de deux vocations distinctes. D'abord, il sert d'intermédiaire entre les étudiants et les enseignants, notamment pour la remise des travaux pratiques de programmation, travaux que nous désignerons sous l'acronyme « TP » (invariable en nombre, même lorsqu'utilisé comme nom). Pour ce faire, un enseignant utilise Oto pour créer une boîte à TP. Celle-ci est assimilable à une véritable boîte matérielle dans laquelle les étudiants peuvent déposer leurs travaux par l'intermédiaire d'Oto. L'enseignant peut ensuite en récupérer le contenu pour corriger les TP qui ont été remis.[2]

Ensuite, Oto sert également à effectuer certaines tâches de correction sur les travaux de manière automatique. Ces corrections peuvent être faites autant par les étudiants que par leur enseignant. Un enseignant peut fournir une évaluation permettant aux étudiants de vérifier une partie du fonctionnement de leurs travaux avant de les rendre et exiger la satisfaction d'une note minimale avant d'accepter la remise. Il pourra utiliser des évaluations plus complexes sur chacun des TP du groupe pour les noter. Une fois la correction terminée, l'enseignant reçoit un rapport comprenant les résultats de celle-ci. Une évaluation consiste en un ou plusieurs fichiers qui permettent de définir une tâche de correction. Cette dernière comprend obligatoirement un fichier décrivant la tâche dans un format qu'Oto comprendra : le script Oto. Le script est en quelque sorte le programme qui sera exécuté lors de la correction et qui manipulera les composants d'Oto pour obtenir le résultat souhaité. Au cours de notre recherche, nous avons été appelés à modifier considérablement la définition et le fonctionnement des scripts, comme

---

[2]Cette fonction de « manipulation de fichiers » remplaçant avantageusement les scripts Unix `rendre_tp/prendre_tp` utilisés à l'UQAM pour la remise des travaux aux enseignants. Ceux-ci étaient problématiques, car la connaissance de ce système n'étant pas nécessaire pour les premiers cours de programmation, il ne pouvait être exigé des étudiants de les utiliser (33).

nous le verrons au cours de ce mémoire.

1.2.1 Commandes Oto

Pour utiliser Oto, l'utilisateur a recours à des commandes sur le shell Unix.[3] Une commande Oto est une série d'instructions permettant de manipuler les boîtes et les évaluations, ainsi que d'effectuer des actions sur celles-ci (par exemple, rendre un TP par la commande `rendre_tp`) et manipuler les fichiers remis. Par exemple, une commande permet à l'enseignant de déplacer dans le répertoire courant les TP qui ont été remis dans l'une de ses boîtes. Une autre commande permet de créer des évaluations qui pourront être associées à une boîte ou être exécutées sur un seul TP ou un groupe entier.

Le nombre de commandes Oto disponibles a augmenté au fil du temps, alors que des besoins nouveaux ont été découverts. Alors que la version 1 ne présentait qu'un ensemble minimal de commandes, les versions suivantes se sont enrichies de commandes diverses permettant notamment de consulter la liste des travaux remis dans une boîte, de lister les caractéristiques sélectionnées à la création d'une boîte, etc. Une liste complète des commandes est présentée à l'appendice A.

1.2.2 Modules d'extension

La généricité et l'extensibilité d'Oto trouvent leur source dans le mécanisme de modules d'extension. Pour que l'outil soit extensible, la mise en œuvre des corrections est découplée de leur coordination. Ainsi, à lui seul, le noyau d'Oto ne dispose d'aucune capacité de correction de travaux. Celles-ci lui sont fournies par des modules d'extension indépendants. Par exemple, Oto dispose depuis le tout début d'un module lui permettant de compiler des programmes Java au moyen du compilateur *javac* et d'un autre lui permettant d'effectuer des corrections introspectives de classes Java avec `JUnit`.

Techniquement parlant, un module d'extension est un fichier Ruby (*.rb*) qui définit plusieurs constantes et une classe contenant des méthodes utilisées par Oto lorsque le module est utilisé dans un script. Pour effectuer leur travail, les modules peuvent accepter des paramètres

---

[3]On peut aussi utiliser Oto à travers son interface Web. Il n'est toutefois pas inexact d'affirmer qu'Oto demeure manipulé à travers la ligne de commande, car c'est en fait ce qui se produit à l'arrière-plan.

à l'entrée et retournent habituellement des résultats. Ces paramètres et résultats doivent être déclarés dans le fichier du module.

Initialement, Oto ne disposait que de deux modules. Ce nombre a été quelque peu augmenté au fil du temps, haussant les capacités d'Oto. Dans le cadre de notre travail, nous avons été appelés à modifier plusieurs éléments des modules pour en augmenter les capacités, notamment en ce qui a trait à la correction intra-groupe pour laquelle Oto n'était initialement pas conçu (17). Nous y reviendrons à plusieurs reprises dans le présent mémoire et, à l'instar des commandes, présenterons une liste de l'évolution des modules à l'appendice A.

1.2.3 Évaluation et script d'évaluation

Derrière la correction automatisée apportée par Oto se trouve l'évaluation. Elle possède un nom unique et peut être privée (accessible par son créateur seulement) ou publique (accessible à tous). Une évaluation se compose de tous les fichiers nécessaires pour effectuer une tâche de correction ainsi que d'un script Oto. Considérons ces éléments en deux phases. D'abord, les fichiers nécessaires pour effectuer une tâche peuvent varier grandement en fonction de la nature de celle-ci. Pour une correction utilisant JUnit, il s'agira de fichiers `.class` définissant les cas de tests.[4] Pour une correction de type filtre, ce sera des fichiers définissant les entrées/sorties à fournir et attendues des programmes à tester.

Le script, à l'opposé, permet véritablement de définir ce que sera l'évaluation en indiquant à Oto quels modules utiliser pour effectuer sa correction et comment agir en fonction des résultats obtenus. À titre d'exemple, on peut inclure dans le script une série de tests hiérarchiques, où les tests plus complets ne sont exécutés que si des tests plus simples réussissent. En ce sens, il s'agit d'un véritable programme qui est exécuté par Oto. Un script doit comporter l'extension `.oto` pour qu'il puisse être identifié à travers les différents fichiers qu'une évaluation peut comporter. Initialement développés dans le langage OtoScript, ils sont à partir d'Oto 2 faits en Ruby, en raison des limites et des problèmes qu'entraînait OtoScript et son support.

---

[4] Et possiblement de fichiers `.java` ou `.class` pour compiler les fichiers fournis par l'étudiant.

1.2.4 Exemple de scénario d'utilisation

À titre d'exemple de ce que nous venons de décrire, considérons la figure 1.1 représentant un scénario d'interaction entre l'enseignant et ses étudiants dans le cadre d'un TP tiré du mémoire de maitrise de Guérin (17). L'enseignant crée d'abord une boîte de remise et une évaluation comportant un certain nombre de tests à effectuer sur les travaux des étudiants. L'étudiant fait ensuite son TP, puis utilise Oto pour le vérifier. Il tient compte des résultats fournis par Oto pour l'améliorer. Il rend ensuite son travail au moyen d'Oto. Une fois que le délai qu'il avait accordé à ses étudiants est écoulé, l'enseignant utilise Oto pour prendre les TP, désactive la boîte de remise et utilise Oto pour effectuer la correction des travaux. Celle-ci sera vraisemblablement plus complète que les tests qu'il avait mis à la disposition des étudiants. L'enseignant peut ensuite nettoyer son espace Oto en désactivant les évaluations qui ne seront plus utiles.

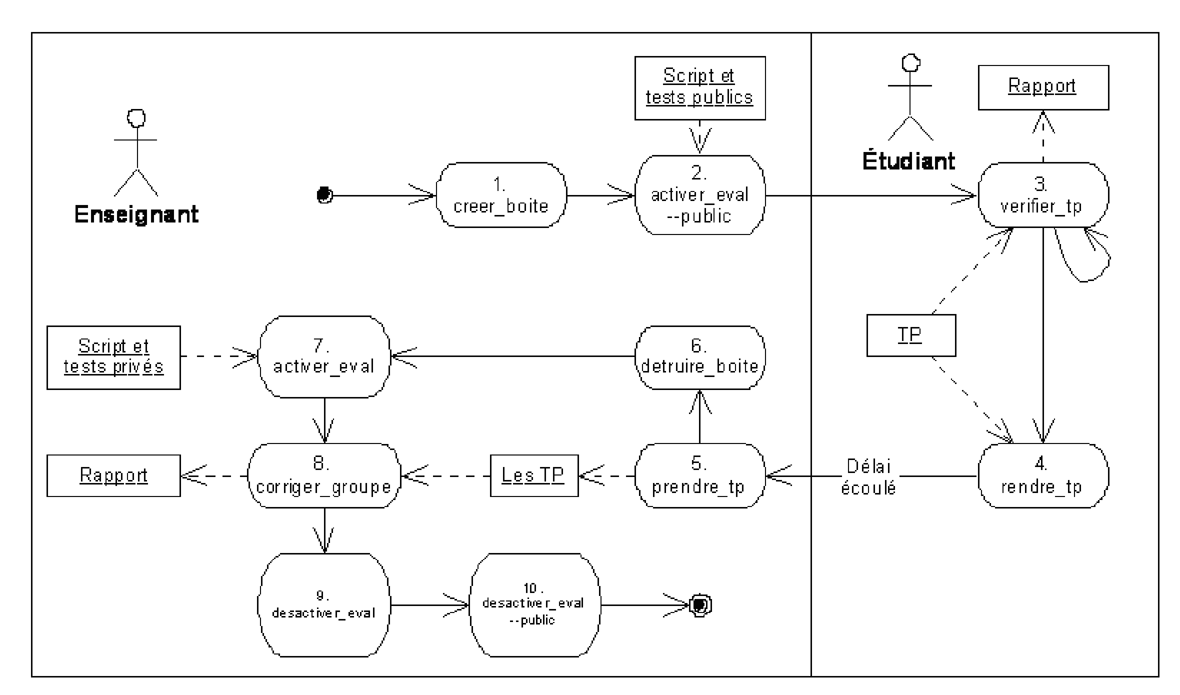


Figure 1.1 Scénario typique d'encadrement d'un TP par Oto (17).

1.3 Les projets liés à Oto

Depuis son lancement en 2005, Oto a été en constante évolution, que ce soit au niveau des commandes, des modules ou des interfaces permettant de l'utiliser. Dans cette section, nous traiterons de certaines des améliorations qui ont été apportées à Oto ainsi que des projets connexes qui ont visé à faciliter son utilisation, tant pour les étudiants que pour les enseignants.

1.3.1 Ajout de modules et de commandes

La version 1 d'Oto ne comportant à l'origine qu'un ensemble réduit de commandes et se limitant à deux modules d'extension, le développement de nouveaux modules est rapidement apparu important aux yeux du professeur Tremblay. Aux modules permettant de compiler des programmes Java et d'effectuer des tests avec JUnit, celui-ci a ajouté :

- Un module de compilation de programmes C (avec le compilateur `gcc`)
- Un module permettant de lancer une commande quelconque sur le shell
- Un module de tests de type filtre sur les entrées/sorties

Le professeur Tremblay a également ajouté des commandes facilitant l'utilisation des tests de type filtre et introspectifs qui permettent de créer automatiquement les tests et les évaluations nécessaires, en une seule commande, en ne fournissant que les entrées/sorties attendues ou le corps des méthodes nécessaires. Il a aussi mis en œuvre des commandes permettant de lancer directement ces tests (34).

1.3.2 Plugin BlueJ

L'outil BlueJ est un environnement de développement intégré (IDE, en anglais) qui est destiné à faciliter l'apprentissage du langage Java en fournissant un environnement de travail simplifié. À l'UQAM, au moment de rédiger ce mémoire, BlueJ était utilisé dans les deux premiers cours de programmation en Java, INF1120 et INF2120. Dans ces conditions, l'idée d'intégrer l'utilisation d'Oto à ce logiciel, facilitant d'autant l'emploi de l'outil dans un contexte de laboratoire devenait intéressante, car elle permettait d'éviter l'apprentissage que demande l'interface ligne de commande.

Malheureusement, bien que BlueJ dispose d'un mécanisme de modules d'extension, ce projet d'extension a été mis de côté en raison de problèmes de coupures intermittentes de la connexion entre les machines locales et le serveur sur lequel Oto était exécuté.

1.3.3 Application Web

L'abandon du projet d'extension Oto pour BlueJ, combinée à une incertitude quant à l'avenir de l'utilisation de cet environnement de développement dans les premiers cours de programmation à l'UQAM, a motivé le développement d'une application Web permettant d'utiliser Oto à partir d'un fureteur au lieu de devoir utiliser son interface ligne de commande. Cette tâche fut confiée à Mohamed Takim, étudiant à la maîtrise en génie logiciel (31). L'application est disponible sur le site d'Oto (`http://www.labunix.uqam.ca/~oto`). Si nous croyons qu'elle a quelque peu facilité l'utilisation d'Oto, particulièrement pour les étudiants, une partie considérable de sa mise en œuvre demeure discutable. Nous en traiterons plus en détails au chapitre 2, et lui consacrons l'appendice C.

1.4 Un module de détection du plagiat

Notre phase d'apprentissage initiale de l'architecture d'Oto complétée, nous avons cherché à parfaire notre compréhension du fonctionnement de ses modules en en développant un nouveau. À ce sujet, les idées ne manquaient pas : détection du plagiat, évaluation qualitative du code source du TP de l'étudiant[5], détection de la présence de certains mots-clefs (comme le `goto`), etc.

C'est finalement sur le premier que notre choix s'est arrêté, car la question du plagiat tenait particulièrement à cœur tant au professeur Tremblay qu'à l'auteur de ces lignes[6], d'autant plus que nous étions convaincu qu'ajouter un module simple à utiliser pour détecter le plagiat intra-groupe dans les travaux d'étudiants deviendrait un argument contribuant à la diffusion d'Oto. L'idée était dès lors de mettre au point un module qui pourrait examiner le code source des TP et mettre en évidence ceux dont la similitude serait suspecte, afin d'inciter l'enseignant

---

[5]Qui serait en mesure de vérifier, par exemple, la présence de commentaires aux endroits appropriés.

[6]Ironiquement, nous avions fait ce choix avant que la question du plagiat ne soit abordée dans les médias montréalais en 2008.

à les vérifier.[7] Il allait de soi que ce module serait essentiellement utilisé dans le cadre de la correction d'un groupe entier (`corriger_groupe`).

La littérature que nous avons consultée au sujet du plagiat dans les travaux de programmation nous a étonné par sa diversité. Pour ne pas alourdir ce chapitre, nous avons préféré placer à l'appendice B un rapport que nous avons rédigé sur le sujet, contenant les principales avenues que pourraient prendre un module de détection du plagiat.

Dès le début, nous avons réalisé que le travail réalisé par un module de détection du plagiat pourrait se diviser en deux tâches distinctes : d'abord, préparer la liste des fichiers à comparer, puis appliquer un algorithme de comparaison sur ceux-ci et en traiter les résultats. La préparation de la liste de fichiers à comparer semblait naturellement relever du module lui-même, lequel pourrait traiter diverses options d'inclusion ou d'exclusion de fichiers, de préparation des fichiers, etc. Toutefois, la mise au point d'un algorithme de comparaison de fichiers, s'il s'agissait d'un sujet de recherche intéressant, semblait éloignée de notre objectif d'améliorer Oto, d'autant plus que nous avons pu trouver plusieurs logiciels dont c'était l'objectif.

Pour cette raison, notre choix s'est finalement arrêté sur une approche ayant recours à un logiciel externe s'occupant de la détection et laissant au module Oto la charge de préparer les fichiers à corriger et traiter les résultats de comparaison.

1.4.1 SIM

Le logiciel que nous avons choisi pour les fins de comparaison se nomme SIM (16). Notre choix a été basé sur plusieurs critères. D'abord, c'était un logiciel à source ouvert, développé en langage C. Ensuite, nous pouvions conserver une copie locale de l'exécutable, contrairement à d'autres outils où les fichiers comparés devaient être transférés sur un serveur à distance qui se chargeait lui-même de la correction.[8] Finalement, SIM pouvait être adapté à plusieurs langages de programmation simplement en définissant les jetons de ces langages, alors que d'autres outils se destinaient à comparer les fichiers développés dans un seul langage cible. Considérant que

---

[7] La correction manuelle étant ce qu'elle est, il serait vraisemblablement difficile pour lui de reconnaître la similitude en cas de faible localité temporelle entre leur correction dans le cadre d'un groupe de taille importante.

[8] Évitant ainsi de mettre Oto et l'UQAM en situation de dépendance à un service fourni par une autre institution d'enseignement ou une entreprise.

nous souhaitions qu'Oto puisse être utilisé dans le cadre d'un maximum de cours d'informatique offerts à l'UQAM, nous ne pouvions nous restreindre à détecter le plagiat dans un seul langage.

1.4.2 Mise en œuvre

La mise en œuvre de notre module de détection du plagiat a repris les deux phases dont nous parlions plus haut. Celle-ci ne s'est pas faite sans heurts (nous verrons pourquoi à la section 2.3.2.3). D'abord, nous traitions les fichiers présents dans les répertoires des étudiants. Cela pouvait se faire de plusieurs manières. Par défaut, le module ne faisait aucune validation : il considérait tous les fichiers de tous les TP du groupe corrigé et les comparait à tous les autres. Nous avons toutefois ajouté certains paramètres, tous optionnels, destinés à augmenter la puissance et la flexibilité du module.

- Permettre de choisir le langage cible parmi une liste prédéfinie. Par défaut, le module s'attend à des fichiers Java.
- Exclure ou inclure des fichiers de la comparaison. Une telle possibilité est utile dans le cas où tous les TP disposeront de certains fichiers communs, comme une classe Java facilitant la manipulation des entrées clavier, telle qu'utilisée dans le cours INF1120.
- Indiquer où seront les répertoires contenant les fichiers devant être comparés au répertoire courant. Une telle manière de faire permettait à un enseignant de se construire une banque de travaux pratiques et de comparer les travaux d'un groupe à cette banque, pour détecter les cas où un étudiant reprendrait un travail remis par un autre à une session précédente.
- Spécifier le plancher au delà duquel le module suspectera un cas de plagiat. La valeur de ce plancher devait être proportionnelle au contenu commun dans un fichier. Normalement, plus le fichier sera grand, plus le plancher pourra être bas.
- Lorsqu'un enseignant remet un ou plusieurs fichiers incomplets aux étudiants pour qu'ils en comblent les vides, une grande partie de ces fichiers sera identique d'un TP à l'autre. Pour que la comparaison se fasse uniquement sur les parties communes, nous avons prévu le concept de `squelette`. Lorsqu'un fichier incomplet est fourni au module, il va utiliser la commande Unix `diff` pour extraire les parties communes des travaux avant de faire la comparaison, évitant des faux positifs.

Le traitement de ces paramètres se faisait avant la comparaison. Celle-ci était faite en fonction du langage cible. Pour permettre une modification de l'algorithme de comparaison sans altérer le reste du module, trois méthodes ont été placées dans le module pour chaque langage supporté, une méthode servant à préparer les fichiers, si un algorithme de comparaison l'exigeait, une autre invoquant l'outil SIM et retournant le résultat obtenu, ainsi qu'une dernière méthode nettoyant les répertoires si nécessaire. En fonction du langage, les méthodes étaient appelées par réflexivité, comme le langage Ruby le permet facilement.

1.4.3 Résultats

Les résultats retournés par le module indiquent, pour les cas suspects, le fichier de l'étudiant concerné ainsi que le répertoire du plagiaire potentiel et le nom de son fichier.

Pour tester notre module, nous avons repris les données de tests provenant de vrais TP remis par des étudiants dont nous avions traité dans notre document préliminaire (voir l'appendice B). Les résultats que nous avions obtenus à l'exécution du module concordaient avec l'utilisation manuelle de SIM que nous avions faite durant notre étude préliminaire.

1.5 Suite de ce mémoire

Le reste de ce mémoire traitera de nos efforts pour améliorer Oto. Après avoir identifié les problèmes de flexibilité et de performances, nous avons voulu en savoir plus sur le fonctionnement interne d'Oto. Pour ce faire, nous avons procédé à une analyse approfondie de l'outil, où nous avons analysé les choix technologiques et de mise en œuvre ayant permis le fonctionnement de ses versions initiales, mais entraîné les problèmes que nous connaissons. À la suite de notre analyse, nous avons émis plusieurs recommandations susceptibles d'améliorer Oto. Celles que nous avons jugées comme étant prioritaires furent mises en œuvre lors du travail qui a conduit à ce mémoire, où la théorie derrière nos choix, leur mise en œuvre et leurs bénéfices par rapport aux anciennes versions, tant au niveau de la flexibilité que de la performance, seront mis en lumière.

CHAPITRE II

PROBLÈMES ET STRUCTURE DES VERSIONS INITIALES D'OTO

Quels sont les problèmes qui ont mené à notre projet de maîtrise? Comment les versions initiales d'Oto étaient-elles mises en œuvre? Comment fonctionnait OtoScript? Comment Oto supportait-il l'exécution de ses évaluations? Quel était le modèle d'exécution des corrections? Pourquoi OtoScript était-il difficile à entretenir? Pourquoi Oto souffrait-t-il de longueurs lors de certaines exécutions? Que devrons-nous faire pour améliorer l'outil?

Dans ce chapitre, nous présenterons d'abord les problèmes ayant mené à nos modifications majeures de l'outil qui feront l'objet des chapitres suivants. Pour identifier les causes de ces problèmes, nous avons procédé à une analyse de l'outil dont nous présenterons les résultats ensuite. Cette étape a pu aboutir sur une série de recommandations de changements dont nous reprendrons les plus importants dans les chapitres suivants. Avant d'en arriver là, nous détaillerons ici certains des aspects techniques les plus importants d'Oto.

2.1 Problèmes des premières versions d'Oto

Malgré leur potentiel intéressant, les versions initiales d'Oto souffraient de défauts importants que nous pouvions soulever à la lumière de l'utilisation d'Oto en situation réelle, notamment au niveau de la flexibilité de son utilisation, tant au niveau des commandes Oto que des scripts, ainsi que de performances relativement faibles. Dans cette section, nous traiterons de ces problèmes à un haut niveau, puis consacrerons le reste de ce chapitre à en chercher la source.

2.1.1 Activation, exécution et désactivation de l'évaluation

Pour exécuter une évaluation, les premières versions d'Oto demandaient que celle-ci soit d'abord activée à l'aide de la commande *activer_eval* où, en plus du fichier *.oto*, les fichiers éventuellement nécessaires à l'exécution de l'évaluation étaient fournis à Oto. Une fois l'évaluation activée, elle pouvait être utilisée avec les commandes *verifier_tp* pour corriger un seul TP ou *corriger_groupe* pour l'appliquer sur un groupe entier. Par la suite, l'évaluation pouvait être désactivée à l'aide de la commande *desactiver_eval*.

Une telle manière de faire était problématique à deux niveaux. Premièrement, elle était lourde, exigeant trois commandes là où l'intention de l'utilisateur était simplement de corriger un groupe. Deuxièmement, une erreur logique dans l'évaluation n'était découverte qu'au moment de la correction, exigeant de la désactiver, puis de l'activer à nouveau après avoir modifié le fichier *.oto*, ce qui exigeait des efforts supplémentaires.

De manière à contourner cette limitation, le professeur Tremblay a eu recours à plusieurs reprises à des scripts qui exécutaient successivement les trois commandes. Cette astuce n'était toutefois pas documentée dans le manuel d'utilisation d'Oto destiné aux enseignants (`http://labunix.uqam.ca:8181/~oto/manuel-enseignant.doc`) et exigeait, vraisemblablement, une connaissance plus que superficielle des commandes Oto. Une meilleure solution aurait été de pouvoir exécuter directement l'évaluation définie dans un fichier *.oto* en fournissant celui-ci et les répertoires des TP à corriger. Nous en traiterons à la section 4.5.1.

2.1.2 Structures de contrôle

Le langage OtoScript, utilisé dans les versions initiales d'Oto pour le développement des scripts définissant les évaluations, était limité à plusieurs niveaux, notamment pour ce qui était des structures de contrôle de l'exécution (l'exécution conditionnelle de code se limitant aux seules assertions, sans boucles et sans structures de branchement). Entre autres, cela rendait difficile la correction à l'aide de programmes de tests multiples et indépendants.

2.1.3 Difficulté à accomplir des tâches d'apparence simple

Parfois, dans le cadre d'une correction, il arrive que l'enseignant doive manipuler les fichiers remis par l'étudiant avant de pouvoir, notamment, les compiler ou leur faire subir des

tests. Par exemple, si les étudiants remettent leurs TP respectifs dans des fichiers compressés, il faudra tout d'abord utiliser des commandes de décompression. Dans le cadre de correction de travaux de programmation, il peut arriver que l'enseignant souhaite écraser certains des fichiers des TP des étudiants par les siens, notamment s'il leur a fourni des fichiers d'interfaces ou de bibliothèques de code que l'étudiant aurait pu modifier malgré la présence d'éventuelles instructions contraires.

De telles tâches, malgré leur apparence simple (puisqu'elles peuvent être accomplies par quelques commandes shell Unix) représentaient une difficulté importante dans les versions initiales d'Oto. Si Oto 1+ disposait d'un module permettant d'exécuter une commande `bash` quelconque, deux défauts demeuraient. D'abord, son utilisation était répétitive, exigeant un appel de module de plusieurs lignes pour chaque commande. Ensuite, même dans un tel cas, il n'était pas évident pour l'enseignant d'inclure dans son script le chemin vers le répertoire contenant les fichiers des TP lors de la correction et de le fournir à l'exécution d'une commande bash.[1]

Remarquons qu'il aurait été possible de créer un module pour effectuer des copies de fichiers, ou un autre pour décompresser des fichiers `.tar.gz` et ainsi de suite. Toutefois, procéder ainsi aurait entraîné une multiplication du nombre de modules nécessaires à l'exécution, alourdissant tant l'apprentissage d'Oto que l'écriture des scripts là où une commande de quelques caractères aurait suffi. De plus, elle aurait rendu la tâche difficile aux utilisateurs d'Oto qui auraient souhaité effectuer une tâche pour laquelle il n'existait pas de module.[2]

2.1.4 Traitement des résultats

Les versions initiales d'Oto étaient peu flexibles au niveau du traitement des résultats : une correction résultait soit en un rapport de correction affiché à l'écran, soit en un message d'erreur. Oto ne permettait pas d'écrire les résultats dans des fichiers sur disque, de les transmettre par un autre moyen, notamment par courrier électronique, ou encore de les remettre aux étudiants. Si une redirection de la sortie standard vers un fichier sur la ligne de commande permettait de

---

[1] Pour éviter d'endommager les fichiers originaux des TP, Oto copie le contenu des TP à corriger dans des répertoires temporaires.

[2] D'autant plus que ces utilisateurs n'auraient pas nécessairement eu le temps ou l'intérêt d'apprendre le fonctionnement des modules Oto pour développer le leur.

contourner le premier problème, le traitement des résultats laissait généralement à désirer, alors qu'il y avait là un éventail de possibilités qui n'étaient pas exploitées.

2.1.5 Performances et lenteurs

Pour qu'Oto puisse être utilisé dans l'ensemble des situations pour lesquelles il avait été prévu, il se devait d'offrir un niveau de performances acceptable. Celui-ci va de quelques secondes pour une simple vérification individuelle à quelques minutes pour la correction d'un groupe.

Selon les observations effectuées par le professeur Tremblay[3], les versions initiales d'Oto pouvaient prendre jusqu'à plusieurs minutes pour retourner les résultats de tests relativement simples. Ces lacunes étaient principalement observées au niveau de tests de filtrage des entrées/sorties, vraisemblablement en raison de la complexité de leur mise en œuvre et du nombre d'éléments impliqués : traducteur de scripts Oto, module Oto en Ruby, scripts `C-Shell`, machine virtuelle Java, etc.

Toutefois, si la question des performances a été en quelque sorte à l'origine de notre projet, nous nous sommes rapidement rendu compte que la flexibilité représentait un problème encore plus important pour la réussite d'Oto lorsque nous avons songé à porter Oto du serveur pour lequel il fut développé, le vieillissant Arabica, vers des machines plus récentes pour, en quelque sorte, contourner les délais observés par le professeur Tremblay. Cette migration sera abordée au prochain chapitre.

Nous ne nous sommes pas toutefois limités à compter sur une amélioration des performances du matériel pour hausser les performances d'Oto. Profitant de la mise en chantier de l'outil afin d'en améliorer la flexibilité, nous nous sommes efforcés de choisir de nouvelles approches moins coûteuses en performances que certains choix qui avaient été faits pour les versions initiales d'Oto.[4]

---

[3]Qui étaient de nature anecdotiques. Toutefois, en l'absence de données d'utilisation d'Oto à grande échelle lors de situations de tests, nous estimons qu'elles furent néanmoins suffisantes pour soulever des doutes quant aux performances de l'outil.

[4]À la défense de ces dernières, remarquons toutefois qu'Oto visait initialement à prouver la faisabilité d'un outil de correction générique et extensible, et non à produire ces résultats en-deçà d'un délai le moindrement raisonnable.

2.2 Analyse et structure de l'application

Ayant identifié certains des problèmes d'Oto, nous avons par la suite cherché à en trouver la cause. Dans cette section, nous traiterons de la structure interne des versions initiales d'Oto, de la disposition de son code dans les répertoires où il est installé et quelques uns des aspects intéressants de son architecture, contribuant à sa souplesse, sa généricité ainsi que son extensibilité, et renforçant ses capacités de correction.

2.2.1 Aspects techniques

Si Oto se présente comme un seul logiciel, il s'acquitte de deux tâches relativement indépendantes : manipuler des fichiers et exécuter des corrections de TP (17). Dans leur mise en œuvre, ces tâches partagent un certain nombre de principes et de mécanismes, notamment des tests unitaires.

Oto est développé en Ruby (32), qui est un langage à objets interprété.[5] L'outil est formé d'un noyau autour duquel viennent se greffer des commandes lui permettant de communiquer avec l'extérieur et des modules d'extension mettant en œuvre ses fonctionnalités de correction.

2.2.2 Chargement anonyme des modules

Le chargement anonyme des modules, décrit par Frédéric Guérin dans son mémoire (17), donne à l'utilisateur d'Oto la possibilité de développer ses propres modules d'extension. Lorsqu'un utilisateur souhaite effectuer une tâche de correction, celle-ci peut utiliser un ou plusieurs modules d'extension Oto qui ne sont pas installés dans les répertoires de l'outil, en autant qu'il les fournisse à Oto comme paramètres à la vérification d'un TP ou à la correction d'un groupe. Cette particularité offre aux correcteurs la possibilité de développer leurs propres modules faits d'une ou de plusieurs classes Ruby, modules qu'ils peuvent utiliser en situation réelle sans privilèges d'administrateur. Or, pour être en mesure d'utiliser une telle classe Ruby, Oto doit l'intégrer à son environnement d'exécution où une classe de même nom existe peut-être. Pour éviter une telle collision, Ruby attribue un nom unique au module.

---

[5] Pour une discussion plus élaborée concernant le choix du langage Ruby, le lecteur se référera à la section 2.3.6.1.

2.2.3 Intégration des tests

L'un des aspects les plus intéressants d'Oto est son mécanisme de tests, lequel permet notamment de vérifier plus facilement la non-régression lors d'entretiens ou d'ajouts à Oto. Ces tests se sont révélés particulièrement utiles lorsque nous avons dû tester notre portage d'Oto vers plusieurs serveurs différents de l'UQAM, car ils ont permis de mettre en lumière plusieurs problèmes qui auraient été difficiles à repérer en l'absence d'un tel cadre.

Oto dispose d'assertions et d'une version modifiée du cadre de tests Ruby *Test/Unit* fonctionnant de manière hiérarchique, c'est-à-dire qu'un cas de test peut exiger la réussite d'autres cas avant de s'exécuter.[6] De plus, Oto fournit des classes destinées à faciliter le travail du développeur de tests en permettant de créer pour lui des répertoires temporaires et en y copiant les fichiers dont il a besoin. Un fichier de tests devrait exister pour chaque fichier `.rb` de mise en œuvre et être placé dans le même répertoire que celui-ci. Mentionnons toutefois que ce mécanisme de tests ne peut évaluer la qualité ou la justesse des tests qu'il exécute. Si les scénarios inclus avec l'application étaient inappropriés ou incomplets, leur utilité serait réduite. L'infrastructure mise en place par Oto permet toutefois au développeur de se concentrer sur les cas à tester en mettant à sa disposition les services nécessaires à leur exécution, ce qui, nous en sommes convaincus, contribue à en hausser la qualité.

2.3 Critique des choix architecturaux et de mise en œuvre

Dans cette section, nous traiterons de certains choix architecturaux d'Oto, particulièrement la mise en place de son environnement d'exécution, le langage OtoScript, les modules d'extension, les rapports de correction et le mécanisme de testabilité. Nous discuterons des stratégies de mise en œuvre utilisées, de leurs forces et de leurs faiblesses.

2.3.1 Le langage OtoScript

Le langage OtoScript était le langage de programmation propre aux versions initiales d'Oto qui servait à coordonner la manipulation des modules d'extension pour effectuer une tâche de correction donnée. Le correcteur pouvait utiliser des constantes, des variables et des

---

[6] Par exemple, les tests avec JUnit exigent le fonctionnement préalable du module de compilation de programmes Java. Ces dépendances ne doivent toutefois pas être circulaires.

assertions simples, utilisables sur les résultats obtenus. Avec ces dernières, le correcteur pouvait hiérarchiser l'exécution de tests dans un même script. Il pouvait également *décorer* un script en ajoutant des étiquettes qui commentaient le rapport de correction.

OtoScript était un langage interprété. À l'activation d'une évaluation, le fichier OtoScript la décrivant était fourni en paramètre. Celui-ci subissait une analyse lexicale, syntaxique et sémantique réalisée par un programme externe. Ce programme, le traducteur Oto, recevait un fichier OtoScript brut et retournait une version épurée appelée image. Cette image était conservée par Oto pour être utilisée lorsque cette évaluation serait invoquée. Lorsque le correcteur effectuait une correction, l'évaluation était chargée et interprétée par un moteur intégré à Oto, qui en analysait l'image et en exécutait les tâches séquentiellement. Cette approche, bien qu'ayant servi les versions initiales d'Oto, n'était cependant pas sans failles.

2.3.1.1 Problèmes liés à l'emploi d'un traducteur

Le traducteur Oto était une application complètement distincte, développée en langage C++, et compilée pour un système d'exploitation et un matériel donné, ce qui limitait la portabilité d'Oto aux environnements pour lesquels le traducteur (et les outils nécessaires à sa génération, notamment ANTLR) pouvaient être portés. Bien que certaines commandes Oto pouvaient fonctionner sans traducteur, celui-ci demeurait incontournable pour activer des évaluations, sauf lorsque cette dernière était faite avec une image déjà traduite.

De plus, utiliser un traducteur rendait difficile l'entretien ou l'amélioration d'OtoScript. Ce problème est apparu lorsque le professeur Tremblay a souhaité ajouter des structures de boucles au langage pour factoriser une partie du code des scripts. Or, pareils ajouts n'étaient pas triviaux, car nous devions d'abord effectuer les ajouts à la grammaire du langage, recompiler et tester le traducteur avant de modifier l'interprète intégré à Oto pour supporter la nouvelle instruction dont des cas de tests appropriés devaient être ajoutés à la banque de l'outil. Qui plus est, ces modifications devaient être faites en l'absence d'une politique de versions permettant d'apparier les fonctionnalités supportées par le traducteur et l'interprète.

2.3.1.2 Affichage du contenu d'une évaluation

Nous avons ajouté à la version 1+ d'Oto la commande `afficher_evaluation` qui permet à l'étudiant d'afficher le contenu d'une évaluation publique. Bien que d'une mise en œuvre simple,

cette commande exigeait qu'Oto conserve la représentation initiale d'un fichier OtoScript en sus de l'image, lorsque la première existait, car elle était plus lisible, ce qui entrait en conflit avec l'idée de ne stocker que l'image « exécutable » d'une évaluation.

2.3.1.3 Emploi d'un langage propre à Oto

Le développement des scripts de correction dans un langage propre à un seul outil demeure discutable. Si cette approche avait permis aux auteurs d'Oto de créer un langage « maison » qui répondait à leurs besoins du moment, il n'en demeure pas moins que celui-ci devait être appris par les utilisateurs souhaitant bénéficier des fonctionnalités de correction d'Oto. Cet effort supplémentaire, applicable à ce seul logiciel, pouvait décourager certains utilisateurs potentiels estimant que les bénéfices apportés par l'outil n'étaient pas suffisants par rapport aux efforts nécessaires à son apprentissage.[7]

2.3.1.4 Syntaxe et particularités d'OtoScript

Même si nous estimons que le langage OtoScript était relativement simple, un certain nombre d'inconstances contribuaient à en alourdir l'apprentissage et l'utilisation en situation réelle. Par exemple, supposons qu'un correcteur souhaitait effectuer un test à l'aide du module `junit`. S'il avait nommé la constante retournée par ce module `resultat`, le nombre d'erreurs survenues durant l'exécution du module aurait été stocké comme attribut de la constante (`resultat.nberreurs`). Si nous supposons maintenant que notre correcteur souhaitait s'assurer que le nombre d'erreurs était égal à 0 à l'aide d'une assertion, il aurait pu le faire par l'instruction `assurer ($resultat.nberreurs == 0)`. Dans ce cas, il pouvait manipuler la valeur directement. La situation n'aurait pas été la même s'il avait souhaité inclure le nombre d'erreurs obtenues dans le rapport de correction à l'aide de l'instruction `sortir`. OtoScript ne lui permettait pas d'utiliser directement la valeur `resultat.nberreurs`. Il devait d'abord assigner cette dernière à une variable, puis fournir cette variable à `sortir`. Cette exigence compliquait l'utilisation d'OtoScript, d'autant plus qu'il était nécessaire de tenter d'activer une évaluation pour qu'un message d'erreur produit par le traducteur puisse signaler un problème syntaxique éventuel.

---

[7] Citons à titre anecdotique une conversation entre l'auteur de ces lignes et Jean Privat, professeur au département d'informatique de l'UQAM. Celui-ci, bien qu'intéressé par les capacités de correction de l'outil, ne l'utilisait pourtant que pour la remise des TP de ses étudiants, pour la raison que nous mentionnons.

Mentionnons également le double usage du signe dollar ($), qui servait à la fois à faire appel au contenu d'une constante et à marquer une expression devant être évaluée par Ruby, deux tâches distinctes qui auraient sans doute gagné à posséder leurs identificateurs propres, pour des raisons de clarté.

Même si ces restrictions étaient mentionnées dans le mémoire de maîtrise de Guérin, elles contribuaient néanmoins à alourdir l'apprentissage et l'utilisation du langage. Or, ces limitations étaient contraires aux objectifs visés par l'emploi d'un langage spécifique au domaine de type externe, où la syntaxe devrait refléter le modèle d'affaires ciblé en évitant les compromis possiblement liés à la mise en œuvre (23). Pour toutes ces raisons, nous pourrions dire qu'OtoScript souffrait des inconvénients des langages externes sans profiter pleinement de leurs avantages.

2.3.2 Problèmes liés aux modules Oto

Les modules d'extension d'Oto sont les composantes de l'application qui lui permettent d'accomplir une tâche de correction. Dans les versions initiales, un module Oto devait être développé en langage Ruby. Lorsqu'Oto devait avoir recours à l'un de ses modules d'extension, tel que spécifié par une évaluation, il le chargeait et lui indiquait où se trouvait le TP à traiter.

2.3.2.1 Conception intrinsèque des modules

Les modules d'extension d'Oto lui fournissent sa grande généricité, découplant complètement le noyau de l'outil de la mise en œuvre des corrections. Toutefois, certains choix des versions initiales d'Oto sont discutables, particulièrement au niveau du modèle d'exécution d'OtoScript et de la conception de leurs modules. Ces versions étaient basées sur un principe que nous qualifierons de correction orientée individu, où chacun des travaux composant un groupe était corrigé l'un après l'autre, de manière purement séquentielle, comme s'il s'agissait de vérifications de TP individuelles. La commande `corriger_groupe` validait les paramètres qui lui étaient passés par l'utilisateur, puis, pour chacun des travaux, le copiait dans un répertoire temporaire, chargeait et interprétait le script Oto correspondant, puis cumulait les résultats obtenus dans un rapport de correction. Une fois tous les travaux traités, ce rapport était affiché sur la sortie standard.

Cette manière de faire était peu efficace et limitait les utilisations possibles d'Oto. Nous expliquerons pourquoi dans les sections suivantes.

2.3.2.2 Surcoûts et problèmes de performances

Dès le début de notre analyse, nous avons constaté que l'exécution orientée individu utilisée par les modules des premières versions d'Oto et coordonnée par le langage OtoScript n'était pas optimale lors des corrections de groupe, puisque plusieurs tâches y étaient dédoublées à chaque exécution, exécutées une fois par copie à corriger : la recherche et le chargement en mémoire des modules, la transmission des paramètres, la validation des paramètres, la manipulation de programmes externes, etc.

Au niveau des paramètres, le fichier OtoScript définissant l'évaluation n'étant normalement pas susceptible d'être modifié au cours de la correction d'un groupe, cette validation ne serait nécessaire qu'à la première copie de ce groupe, et pourrait être considérée acquise pour les autres. De plus, il serait souhaitable d'optimiser les appels vers des programmes externes. Un exemple simple est la correction de travaux en langage Java à l'aide de `JUnit` (5) et du module éponyme. Pour chacun des travaux d'un groupe, le module doit charger et décharger la machine virtuelle Java, alors qu'il serait plus intéressant de ne la démarrer qu'une fois et d'effectuer les corrections en lot, ce que les versions initiales d'Oto ne permettaient pas.

2.3.2.3 Difficultés liées aux corrections « de groupe »

Dans les versions initiales d'Oto, la correction se faisait sur un seul répertoire temporaire à la fois. Les modules `javac`, `junit`, `gcc`, `cmd_bash` et autres traitaient tous les fichiers d'un répertoire et généraient un résultat qui ne concernait que le seul TP correspondant. Or, le module `plagiat` que nous avons développé devait comparer les travaux entre eux, ce qui ne pouvait être fait proprement dans les versions initiales d'Oto, car le module ne pouvait connaître le nom véritable du répertoire anonyme qu'il manipulait pour éviter de le comparer avec lui-même (et provoquer des faux positifs).

Pour contourner ce problème, nous avions modifié la commande `corriger_groupe` et lui avions fait générer un fichier texte comportant le nom réel du répertoire actuellement corrigé, qui était placé dans le répertoire courant où le module de détection du plagiat pouvait le récupérer. Cette solution temporaire n'était pas sans défauts, car elle empêchait l'exécution simultanée de plusieurs instances d'Oto depuis un même répertoire (le fichier devant conserver un même nom). De plus, en cas de « plantage » du module ou de la commande, celui-ci n'était pas supprimé. Pire encore, la commande ne pouvant savoir si le module `plagiat` était utilisé ou non

pour la correction en cours, elle devait appliquer ce correctif pour tous les appels de modules, provoquant des surcoûts.

2.3.2.4 Formatage des corrections « de groupe »

Un autre irritant que le module `plagiat` nous a fait découvrir est l'impossibilité, dans les versions initiales d'Oto, de faire des regroupements de résultats dans les rapports. Ceux-ci n'étaient que des amalgames de résultats individuels présentés séquentiellement, sans possibilités de faire des regroupements en fonctions des résultats obtenus. Par exemple, dans le cas de la détection du plagiat, il eut été souhaitable de ne générer qu'une seule liste de tous les travaux similaires.

2.3.3 Problèmes liés aux rapports de correction

Un rapport de correction est le compte-rendu de l'exécution de l'évaluation dans le cadre de la correction d'un TP. Dans les versions initiales d'Oto, ce rapport était produit par l'interprète lui-même. Plusieurs facteurs influençaient ce qui était inclut dans un rapport : la visibilité des tâches de correction, les décorations et les valeurs explicitement affichées à l'aide de l'instruction `sortir`. Cette correction se faisait un seul TP à la fois, un rapport ne pouvait s'appliquer qu'à ce seul TP, même dans le cas d'une correction de groupe auquel seul un bref sommaire était ajouté, informant l'utilisateur de statistiques superficielles liées aux résultats du groupe.

L'idée de retourner les résultats d'une correction sous forme de rapport nous semble bonne. Toutefois, leur mise en œuvre s'avérait discutable, notamment lorsque nous avons cherché à restreindre la quantité d'information retournée par un rapport. Il est important de remarquer que si Oto n'éprouvait aucune difficulté à traiter l'information contenue dans les rapports, c'est parce qu'il les manipulait de manière structurée (par les valeurs de retour des modules d'extension et les constantes des scripts). Pour Oto, un rapport n'était qu'un compte-rendu n'ayant rien à voir avec la réussite ou l'échec d'un script. Finalement, les rapports d'Oto étaient peu adaptés à la lecture logicielle de leurs résultats, ce qui limitait les possibilités de leur formatage par une application externe. Si le rapport de correction était relativement facile à lire par humain, il n'en était pas de même pour les programmes externes désirant manipuler ces résultats, en l'absence d'un standard de nomenclature d'affichage entre les modules et, surtout, d'un balisage lisible par une machine.

L'application Web d'Oto (dont nous traiterons plus en détails à la section 2.5) souffrait clairement de cette incapacité à traiter les résultats d'une évaluation par la machine, car le rapport était présenté sous forme de texte brut, non formaté. Aucune information sur la vérification n'est présentée en dehors de ce champ, alors qu'une présentation qui aurait exploité les capacités inhérentes à l'emploi du HTML ou du XML par l'application Web (listes, tableaux avec niveaux que l'utilisateur peut dérouler ou replier) aurait été plus claire.

2.3.4 Portabilité

La portabilité d'Oto est un sujet important, car l'outil se veut une approche générique et adaptable au plus grand nombre possible de situations de corrections. Au cours de notre recherche, nous avons été amenés à porter l'outil, développé sous Solaris, vers plusieurs machines de l'UQAM disposant de différents environnements de type Unix. Ce portage ne s'avéra pas sans difficultés. Dans cette section, nous examinerons les difficultés liées à un portage sous environnement Unix, puis nous nous demanderons s'il serait possible de le faire fonctionner sous d'autres familles de systèmes d'exploitation.

2.3.4.1 Passage d'un environnement Unix à un autre

Au niveau du code source d'Oto, le passage d'un « Unix » à un autre ne nous a pas posé de problèmes majeurs. La principale difficulté venait d'appels systèmes faisant référence à des commandes et qui étaient codés en dur. À ce niveau, les commandes de type « Créer et activer évaluation », faisant appel à des scripts `C-Shell`, durent être modifiés. Ceux-ci n'ayant sans doute pas été conçus dans l'optique d'être portés, leur code était peu factorisé et supposait qu'une instance d'Oto était accessible dans le `PATH`. Nous avons clarifié le code et passé en paramètre le chemin absolu vers l'instance d'Oto à invoquer pour effectuer leurs tâches. Un script destiné à chaque serveur fut créé. L'environnement Oto indiquait alors aux commandes et aux modules quel script utiliser, ce qui limitait la duplication du code des premières et simplifiait l'entretien des scripts.[8]

[8] Certes au prix de la duplication du code des scripts, mais procéder ainsi réduisait le nombre total de modifications en n'altérant pas les paramètres passés aux scripts et en évitant d'ajouter des conditions un peu partout dans ceux-ci.

2.3.4.2 Autres systèmes d'exploitation

Porter Oto vers un autre système d'exploitation qu'Unix serait beaucoup plus difficile que ce que nous avons réalisé pour Oto 1+. La sécurité de l'outil repose sur les permissions Unix. Oto doit avoir un niveau de permission plus élevé pour manipuler les répertoires des correcteurs, mais bascule vers le niveau de permissions du correcteur lors d'une correction, pour des raisons de sécurité. L'outil est un programme de type SGID, un concept Unix. Aussi, la classe `Usager`, selon les commentaires de son code, représente un utilisateur Unix. D'ailleurs, les tests qui lui sont associés ne s'appliquent qu'à cette seule famille de systèmes d'exploitation, faisant référence à un administrateur se nommant « root ».

2.3.5 Internationalisation

Oto est un logiciel unilingue francophone. Si elle n'est pas problématique à l'UQAM, cette situation diminue les possibilités d'utilisation de l'outil à une plus grande échelle, notamment dans les établissements d'enseignement anglophones. Pourtant, il serait relativement simple de relocaliser l'outil à d'autres langues. Des paquetages de langues pourraient être distribués avec Oto et choisis à l'installation. L'internationalisation ne se limite toutefois pas à la langue, mais touche aussi plusieurs aspects culturels. Par exemple, l'ordre variable des informations dans une date, pourrait être mise en œuvre par le paquetage et modifiée sans altérer les commandes.

2.3.6 Performances

Oto doit supporter plusieurs circonstances d'utilisation : par un enseignant pour créer des boîtes et prendre des TP, par plusieurs étudiants pour vérifier leurs travaux lors d'un laboratoire, par un correcteur corrigeant un groupe, etc. La performance attendue varie en fonction de l'utilisateur. Si un délai de quelques minutes lors d'une correction de groupe ne nuit pas à la pertinence des résultats, une attente aussi longue risque de nuire à un étudiant au cours d'un laboratoire pratique limité dans le temps.

Dans le cadre de notre analyse de l'outil, nous avons cherché à repérer les causes des délais qui ont parfois été observés dans ses réponses. Dans les sections suivantes, nous traiterons d'aspects que nous avons jugés problématiques dans les versions initiales d'Oto. Nous consacrerons la section 2.4 à une analyse dynamique de l'outil qui avait confirmé plusieurs de nos soupçons.

2.3.6.1 Choix du langage Ruby

Le langage Ruby demeure considéré comme l'un des langages les moins performants[9], sinon le moins performant disponible sur le marché. Sans nier ces informations, nous n'avons pas estimé qu'il était nécessaire de réécrire Oto dans un langage réputé plus rapide (par exemple, le langage C), et ce pour plusieurs raisons.

Premièrement, Oto étant déjà développé en Ruby, traduire une grande partie de son code source vers un nouveau langage aurait demandé des efforts peu intéressants dans le cadre de la maîtrise en informatique. D'ailleurs, seuls certains cas d'utilisation d'Oto devaient être complétés rapidement, et même pour ceux-ci, certains modules étaient plus lents que d'autres. Deuxièmement, l'UQAM disposant de meilleur équipement exécutant Oto beaucoup plus rapidement que le serveur Arabica, une telle traduction devenait moins nécessaire.

2.3.6.2 Difficultés liées au mécanisme des modules

Comme nous en avons déjà fait mention, les versions initiales d'Oto appuyaient leur mécanisme de correction sur un langage interprété et des modules orientés individu. Ces facteurs contribuaient à la lenteur de l'application, laquelle s'accroissait à un rythme proportionnel à la taille des groupes corrigés. Les mécanismes des versions initiales d'Oto n'étaient pas conçus dans une optique de performance.[10]

2.4 Profilage et mesure des performances

Dans les sections précédentes, nous avons mentionné que les principaux problèmes de performance d'Oto étaient causés en majeure partie par le paradigme orienté individu sur lequel le langage OtoScript reposait et par le fonctionnement même des modules d'extension. Pour corroborer ces assertions, nous avons analysé le comportement d'Oto à l'exécution à l'aide du profileur *ruby-prof*.[11] Nos essais furent réalisés sur notre poste de travail au LATECE, sous

[9]Selon Tim Bray, directeur chez Sun Microsystems, à la *Silicon Valley Ruby Conference* (`http://www.theregister.co.uk/2008/04/21/bray_ruby_rails/`) en avril 2008.

[10]Bien que nous estimions que notre critique soit légitime, la priorité des premières versions d'Oto était de prouver la faisabilité de leur approche générique, et non la performance, comme nous l'avons déjà remarqué.

[11]`http://ruby-prof.rubyforge.org/`

Linux (Ubuntu 8.04). Remarquons que les performances de ce poste récent étaient telles que les temps d'exécution obtenus furent souvent inférieurs aux délais des serveurs de l'UQAM pour la complétion des mêmes tâches.

2.4.1 Profilage

En premier lieu, nous avons effectué le profilage de l'exécution de tous les tests d'Oto, lancés au moyen de la commande `tester_module` employée seule. Il aurait été possible de spécifier un test ou un ensemble de tests en spécifiant leur nom en paramètre à la commande.

Les résultats obtenus nous ont permis de faire deux constats intéressants. D'abord, Oto passe la majeure partie de son temps à appeler des programmes externes, soit plus de 63 %. Ensuite, le reste du temps d'Oto était divisé entre différentes méthodes : manipulation de chaînes de caractères, lecture de fichiers, manipulation de fichiers et de répertoires, etc. Ces méthodes provenaient surtout de l'interprète des images OtoScript ainsi que du mécanisme des modules d'extension. Quelques unes d'entre elles, bien qu'utilisées un nombre réduit de fois, prenaient plusieurs secondes chacune à s'exécuter, devenant autant de points névralgiques.

Par la suite, nous avons souhaité vérifier si le cadre de corrections de travaux nous permettrait d'en arriver aux mêmes conclusions. Pour ce faire, nous avons limité le profilage d'Oto aux tests automatisés des modules `gcc`, `javac`, `junit` et `plagiat`. Les résultats obtenus pour ces tests confirmèrent les premiers constats, le pourcentage de temps consacré à l'exécution de programmes externes atteignant plus de 75 %. Nous n'avons pas été surpris de tels résultats, car ces modules étaient essentiellement construits pour manipuler des outils externes mettant leurs fonctionnalités au service d'Oto.[12]

Bien qu'intéressants, ces résultats provenaient des tests automatisés d'Oto, et non d'une situation de correction authentique. Pour confirmer que le comportement d'Oto demeure le même dans un tel cas, nous avons effectué la correction d'un groupe de 32 travaux d'étudiants qui ont été comparés avec le module `plagiat`. Non seulement ce test a-t-il permis de confirmer les résultats des profilages mentionnés précédemment, mais il a mis en lumière l'importance d'optimiser le fonctionnement des commandes Oto, car la commande `corriger_groupe` perdait

---

[12] Mentionnons également que le module `plagiat` s'est distingué en effectuant un nombre élevé de manipulations de répertoires, ce qui est caractéristique de son comportement « de groupe » qu'Oto 1+ supportait mal.

du temps à appliquer le correctif permettant d'indiquer au module le nom du répertoire en cours de correction (voir la section 2.3.2.3).

Les résultats détaillés du profilage de la version 1+ d'Oto ainsi qu'une discussion plus approfondie des résultats obtenus sont présentés à l'appendice D.

2.4.2 Performances comparatives selon la version de Ruby

Initialement, Oto était conçu pour des versions relativement anciennes du langage Ruby, soient 1.8.1 et ultérieures (17). Or, au moment de faire notre analyse, alors que les serveurs de l'UQAM disposaient de la version 1.8.6, la version 1.9.0 de Ruby était disponible à titre expérimental, promettant une hausse des performances du langage. Souhaitant vérifier si cette hausse serait bénéfique à Oto, nous avons adapté celui-ci pour qu'il soit utilisable avec cette nouvelle version, ce qui a demandé des changements somme tout mineurs[13], avant de lancer des essais comparatifs présentés au tableau 2.1. Les délais indiqués sont en secondes, le temps le plus rapide de chaque test étant en caractères gras.

Test	**Ruby 1.8.6**	**Ruby 1.9.0**
Démarrage d'Oto sans arguments	**0,10**	0,16
Test de la commande `rendre_tp`	**74,87**	74,90
Test du module `javac`	30,44	**30,05**
Test du module `junit`	129,31	**129,00**
`verifier_tp`, `javac` et `junit`	1,05	**1,03**
`corriger_groupe`, `javac` et `junit`	19,40	**18,47**
`corriger_groupe` et `plagiat`	77,45	**18,84**

Tableau 2.1 Temps d'exécution d'Oto selon la version de Ruby.

À la lumière des résultats que nous avions obtenus, la nouvelle version de Ruby n'apporta généralement que des accélérations modestes, et s'était même avérée légèrement plus lente pour les premiers tests. Cependant, la majorité des résultats étaient serrés, avec un certain avantage

[13]Nous avons essentiellement clarifié des ambiguïtés causées par du *shadowing* de variables locales et corrigé des erreurs de sécurité causées par des données souillées. Le *shadowing* se produit lorsqu'une variable locale possède un nom identique à une variable visible de même nom située dans un bloc englobant.

pour Ruby 1.9.0 lors des corrections de groupes. Considérant ces écarts faibles, le résultat du dernier test fut surprenant, car la version la plus récente du langage n'a pris que 24,33 % du temps de la plus ancienne pour compléter son exécution.

Un tel écart tend à nous fournir une explication sur les causes possibles d'un tel gain. Une meilleure gestion des *hashes*, de la classe `Array`, des chaînes de caractères et des caches serait plausible, car l'emploi du module `plagiat` utilise abondamment les premières. De plus, l'utilisation d'une machine virtuelle par Ruby 1.9.0 contribue possiblement à accélérer le support de la réflexion avec laquelle ce même module lance ses traitements à raison de trois appels introspectifs par exécution réussie de `plagiat`.

2.5 Faiblesses de l'application Web

L'application Web d'Oto, dont nous avons déjà fait mention au chapitre 1, mérite à notre avis elle aussi d'être critiquée, tant au niveau de sa mise en œuvre que de ses caractéristiques non fonctionnelles, notamment la maintenabilité, la testabilité et l'utilisabilité. À notre avis, cette application représente l'un des maillons faibles d'Oto, sa qualité nous ayant apparu généralement fort discutable. Par ailleurs, pour ne pas alourdir ce chapitre, nous avons préféré transposer cette discussion à l'appendice C, où nous en traiterons plus rigoureusement en effectuant notamment une véritable analyse heuristique de son interface pour en vérifier l'utilisabilité. Pour réduire ses défauts, nous avons cherché à améliorer l'interface Web. Par manque de temps, nous avons dû nous contenter de simplifier la page d'accueil en supprimant des éléments décoratifs inutiles et l'avons rendu plus lisible en en haussant la taille de la police de caractère des libellés. Comme nous le mentionnons dans l'appendice, nous croyons que l'application nécessite une refonte majeure.

2.6 Conclusion

Au cours de ce chapitre, nous avons examiné la conception et la construction d'Oto. La cause du manque de flexibilité des scripts de correction a été identifiée. Nous avons également constaté que les lenteurs d'Oto étaient provoquées par un ensemble de choix de mise en œuvre. Nous présenterons maintenant certaines solutions envisagées pour le développement d'Oto 2. Elles se divisent en trois catégories, de nécessaires à souhaitables.

2.6.1 Améliorations nécessaires

Les améliorations que nous avions jugées nécessaires furent celles susceptibles d'apporter des solutions concrètes aux problèmes les plus sérieux d'Oto. Ces améliorations furent celles que nous avons effectuées au cours du reste du travail ayant mené au présent mémoire.

2.6.1.1 Meilleure coordination des tâches

L'utilisation du langage OtoScript pour coordonner les corrections avait montré ses limites. OtoScript souffrait de limitations au niveau de sa syntaxe et péchait par un excès de simplification. Sa maintenance exigeait beaucoup d'efforts : il était nécessaire de maintenir son traducteur, son interprète et leurs tests unitaires.

De manière générale, le langage de coordination des tâches d'Oto devait permettre à l'utilisateur de préparer un ou plusieurs TP en vue d'une correction, d'effectuer une ou plusieurs tâches de corrections sur les TP et de traiter les résultats obtenus. La préparation et la correction peuvent consister en des tâches relativement complexes gérées par des modules ou de simples manipulations qui peuvent être accomplies par une ou plusieurs commandes *bash*. Le traitement des résultats peut varier de l'affichage d'un rapport complexe à l'écran à l'écriture de celui-ci sur disque ou de l'utilisation des données à d'autres fins, telles que la transmission par courrier électronique ou à une application permettant aux étudiants de consulter leurs notes.[14]

Oto étant développé en langage Ruby, l'emploi de ce langage ouvre la porte à l'élimination du traducteur, de l'interprète et d'OtoScript lui-même. Ce dernier sera avantageusement remplacé par un langage spécifique au domaine de type interne (23; 29) s'appuyant sur Ruby.

Les caractéristiques recherchées par ce nouveau langage sont détaillées à la figure 2.2, en comparaison de celles des versions initiales d'Oto. Nous traiterons plus en détails des langages spécifiques au domaine au chapitre 3 de ce mémoire. Notre solution et sa mise en œuvre seront l'objet du chapitre 4.

---

[14]Comme, par exemple, les applications Résultats (`http://www.resultats.uqam.ca/`) et Moodle (`http://www.moodle.uqam.ca/`) utilisées à l'UQAM.

Caractéristique	**Oto 1+**	**Oto 2**
Paradigme de fonctionnement	Un seul TP considéré à la fois	Correction de tout le groupe à la fois
Chargement et exécution du script de correction	Une fois par TP	Une fois pour tout le groupe
Fonctionnement des modules	Individuel seulement	Individuel et collectif
Paramètres d'un module	Tous de type `String`	Objets de divers types
Valeur de retour d'un module	Structure Ruby	Objet
Principe des rapports	Un seul type de rapport codé en dur, non paramétrable	Modulaires et paramétrables
Affichage du rapport	Obligatoire, à la fin de la correction	Facultatif, à n'importe quel moment du script
Possibilité d'écrire le rapport sur disque	Non, sauf en redirigeant la sortie standard à l'appel d'Oto	Oui, directement dans le script
Positionnement des résultats des modules « collectifs » dans le rapport	Avec chacun des TP	Séparément et/ou avec chacun des TP
Contrôle du flux d'exécution	Par assertions seulement	Tests et exécution conditionnelle, boucles, blocs, assertions, etc.
Support d'appel de commandes `bash`	Par un module	Directement dans le script, sans appel de module
Paradigme du langage	Déclaratif à la `makefile`	Procédural, à objets
Développement de classes et de méthodes dans le script	Non	Oui

Tableau 2.2 Caractéristiques et comportement selon la version d'Oto.

2.6.1.2 Refonte du mécanisme de modules Oto

Notre profilage d'Oto nous a permis de confirmer nos soupçons au niveau du comportement des modules. Ceux-ci consacraient beaucoup de temps à agir à titre d'interface avec des applications externes. À ce niveau, nous pouvons imaginer l'obtention de gains de deux manières différentes. En premier lieu, les modules comparant les travaux entre eux doivent intrinsèquement s'appliquer à tous les travaux d'un groupe, au lieu de chaque travail à la fois, afin de pouvoir s'adapter à toute approche souhaitable pour les corrections. Procéder ainsi nous permet de factoriser les tâches qui, anciennement, étaient répétées inutilement. De plus, les appels à des programmes externes doivent s'appliquer à tous les travaux d'un groupe lorsque cela est possible, afin de réduire leur nombre et d'éviter leurs chargements et déchargements répétitifs en mémoire.

Notre solution se tournera vers le développement de deux types de modules, un premier qui s'applique à un seul TP et un autre traitant un groupe entier. Les nouveaux modules, adaptés à la coordination des tâches basée sur un langage interne, seront présentés à la section 4.3 de ce mémoire.

2.6.1.3 Rapports de correction

Les rapports de correction fournis par Oto permettent d'exprimer les résultats obtenus à l'exécution d'une évaluation. Ceux-ci souffraient de plusieurs défauts : ils étaient peu paramétrables, inflexibles et difficiles à analyser par la machine.

Le rapport de correction d'Oto 1+ ne convenait plus. Son contenu devait pouvoir être choisi par le développeur d'une évaluation. De plus, le rapport ne devait plus se limiter à un affichage sur la sortie standard et devait pouvoir, par exemple, être envoyé par courrier électronique. Un « rapport » devait également pouvoir ne rien afficher à l'écran, mais copier ses résultats dans un entrepôt externe. Les résultats de certains rapports devaient également pouvoir être formatés en XML pour être lisibles par la machine. Pour ce faire, nous avons imaginé un mécanisme de « modules-rapports » similaire aux modules d'extension. Les rapports feront notamment l'objet du chapitre 4 de ce mémoire.

2.6.2 Améliorations souhaitables

En plus de celles que nous avons effectuées dans le cadre de notre travail, les améliorations que nous qualifions de souhaitables contribueraient à la diffusion d'Oto. Le remplacement de l'application Web fait également partie de cette catégorie, car la version actuelle, si limitée qu'elle soit, demeure compatible avec la majeure partie des modifications jugées nécessaires que nous avons effectuées.

2.6.2.1 Utilisation de Ruby 1.9.0

Les changements nécessaires à Oto pour qu'il s'exécute sous Ruby 1.9.0 sont minimes. Par ailleurs, les tests que nous avons effectués sous cette version nous ont permis de constater que celle-ci provoque en général une légère hausse des performances de l'application. Seul le module `plagiat` a bénéficié d'une accélération importante. Malgré ces résultats encourageants, lorsque nous avons tenté, par la suite, de porter Oto vers Ruby 1.9.1, version qui a supplanté 1.9.0, de multiples problèmes sont apparus, notamment au niveau des tests et du chargement des fichiers, cette version modifiant le comportement de plusieurs aspects de Ruby.[15] Résoudre ces problèmes exigerait un effort important, alors qu'il s'agit encore d'une version expérimentale de Ruby. Un tel effort nous apparaît peu intéressant, préférant nous concentrer sur le problème de la flexibilité. Pour cette raison, nous préférons attendre que cette nouvelle version de Ruby se soit stabilisée avant de recommander d'adapter Oto pour celle-ci.

2.6.2.2 Portabilité

Dans ses versions initiales, Oto dépendait de l'environnement Solaris de la machine Arabica sur lequel il a été développé. Si nous l'avons porté vers d'autres environnements Unix, il demeure dépendant de cette famille de systèmes d'exploitation. Il serait éventuellement possible de modifier Oto pour qu'il puisse être utilisé sur n'importe quel système d'exploitation disposant d'une version des environnements d'exécution dont il a besoin (Ruby, Java, et autres). Elle nécessiterait de modifier la partie « gestion des fichiers » d'Oto pour qu'elle fonctionne à l'aide d'une base de données pour cesser de dépendre de la sécurité d'Unix, de rendre plus souple l'application de règles lors de la remise de fichiers et de faciliter les scénarii de correction

---

[15] Entre autres, l'instruction `require` qui permet de charger un fichier Ruby souffre d'un problème au niveau du nom des fichiers, soulevant une exception de sécurité si le chemin fourni n'est pas absolu.

multi-enseignants.

2.6.2.3 Internationalisation

Pour l'instant, Oto a toujours été unilingue francophone. Faciliter l'internationalisation d'Oto semble utile. Au delà de la langue, l'internationalisation traite de tous les aspects culturels du logiciel, notamment le format des dates. Mettre au point un système de paquetages permettant l'internationalisation d'Oto permettrait de distribuer l'outil avec plusieurs choix de langues où l'une d'entre elles serait choisie à l'installation.

2.6.3 Améliorations possibles

Les améliorations possibles sont des idées générales de commandes, de modules et d'autres aspects qui seraient susceptibles d'ajouter des fonctionnalités intéressantes à Oto. Toutefois, comme celles de la section précédente, elles ne faisaient pas partie de nos objectifs principaux. Leur mise en permettrait de hausser encore davantage les capacités d'Oto et de rendre son utilisation à une plus grande échelle de plus en plus attrayante en dehors du contexte de l'UQAM. Comme idées possibles, mentionnons une commande qui permettrait d'importer des travaux remis à une adresse de courrier électronique ou par un logiciel d'aide à l'enseignement tel que Moodle, le développement de nouveaux modules Oto, notamment au niveau de la correction qualitative du code, la transmission des résultats directement aux étudiants et la mise en place d'une politique de versions à partir d'Oto 2. Finalement, comme nous l'avons mentionné à la section précédente, une refonte du modèle de stockage où les travaux sont conservés dans des répertoires reposant sur la sécurité Unix par une base de données pourrait augmenter la sécurité de l'application, notamment en spécifiant des listes d'accès limitant quels comptes étudiants peuvent déposer des travaux dans une boîte et en hausser les capacités, facilitant notamment la détection du plagiat lorsque la remise d'un TP identique est exigée simultanément des étudiants de plusieurs groupes d'un même cours.

CHAPITRE III

LANGAGES SPÉCIFIQUES AU DOMAINE

Au-delà du paradigme utilisé, quelles sont les familles de langages de programmation? Quels sont les avantages et les inconvénients des différentes solutions possibles au niveau des langages pouvant être utilisés pour remplacer OtoScript? Qu'est-ce qu'un langage spécifique au domaine (DSL)? Depuis quand existent-ils? Quels sont les différents types de langages spécifiques au domaine? Pourquoi le langage Ruby est-il utilisé pour mettre en œuvre des langages spécifiques au domaine? Quelles sont les caractéristiques de Ruby utilisées à cet effet?

Lorsque nous avons pris la décision de remplacer le langage OtoScript par une solution plus flexible et plus efficace, il nous a été nécessaire de choisir quel langage serait voué à le remplacer. Le professeur Tremblay nous a alors suggéré l'utilisation d'un langage spécifique au domaine basé sur le langage Ruby, qui, selon lui, était une approche de plus en plus utilisée depuis quelques années. Toutefois, avant de traiter des solutions propres à Oto, il nous est apparu important d'en apprendre davantage sur la question de ces langages.

Dans ce chapitre, nous traiterons des langages de programmation en général, et des langages spécifiques au domaine en particulier. Nous verrons les différences entre ces derniers et les langages généralistes, tels que Java ou C++. Nous examinerons ensuite les deux familles de DSL, lesquels sont divisés en fonction de leur mise en œuvre, et discuterons des avantages et inconvénients des deux approches. Finalement, nous discuterons de l'utilisation du langage Ruby dans les DSL. Nous verrons les caractéristiques rendant celui-ci intéressant pour le développement de tels langages et considérerons un exemple disponible sur le Web.

3.1 Langages généralistes et langages spécifiques

Les langages de programmation peuvent, au-delà du ou des paradigmes supportés par chacun, être classés dans deux grandes catégories, soient les langages généralistes et les langages spécifiques au domaine (28; 29). Ces deux catégories de langages se distinguent par leurs objectifs généraux.

Un langage de programmation généraliste est un langage permettant de mettre en œuvre une vaste gamme de tâches dans diverses applications. Du point de vue génie logiciel, sa syntaxe, ses concepts et ses capacités sont génériques. Par exemple, le langage Java possède un grand nombre de classes permettant d'effectuer des tâches aussi diverses que de construire et d'afficher une interface graphique, transmettre des paquets sur les réseaux, se connecter à des bases de données pour échanger avec elles, utiliser des services de noms, émettre du son, etc. Un tel langage ne vise pas à s'adapter à un domaine d'affaires particulier, mais plutôt à satisfaire une vaste gamme de besoins distincts. Sa syntaxe reflète ce généralisme, Java ne disposant pas de mots-clefs propres à un domaine particulier.

Un langage spécifique au domaine, à l'opposé, est conçu spécifiquement pour le domaine d'affaires auquel il servira (29). Comme l'écrit Martin Fowler :

> *The basic idea of a domain specific language (DSL) is a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem.*[1]

En cédant une partie de sa généralité, un langage spécifique au domaine permet de gagner en expressivité au niveau de son domaine cible (23) en baissant son niveau d'abstraction par rapport au langage généraliste, le langage spécifique au domaine possédant des mots-clefs et étant structuré pour représenter les concepts du domaine d'affaires. Utilisé dans le domaine pour lequel il est conçu, un DSL entraîne des gains de productivité de la part de ses utilisateurs. Un niveau d'abstraction plus élevé diminue le temps nécessaire au développeur pour comprendre l'intention derrière le code (13). Les DSL réduisent les coûts d'entretien du code par rapport à un langage généraliste, pouvant utiliser des notations et des concepts propres à leurs domaines (23). Ils permettent également d'exprimer la résolution d'un problème dans une langue

---

[1] `http://martinfowler.com/bliki/DomainSpecificLanguage.html`

que les spécialistes du domaine sont en mesure de comprendre. Considérant la difficulté que représente la communication entre les développeurs et les spécialistes du domaine dans le cadre d'un développement logiciel, l'emploi d'un DSL représente un avantage non négligeable (13).

3.2 Langages spécifiques au domaine

3.2.1 Historique des DSL

L'idée de langages spécifiques au domaine n'est pas récente. Selon Martin Fowler, celle-ci serait presque aussi ancienne que les langages de programmation eux-mêmes. Le concept serait en effet apparu dans la seconde moitié des années 1950 avec APT, un langage visant la programmation de machines-outils (23).

Une difficulté de décrire l'histoire des DSL est qu'il n'y aurait pas unanimité quant à savoir ce qui fait d'un langage un DSL. Selon Mernik, Heering et Sloane (23), la situation ne serait pas noire ou blanche. Les langages pourraient plutôt être classés sur une échelle allant de « très spécifique au domaine » (par exemple, selon les auteurs, la forme de Backus-Naur[2]) à « très généraliste » (par exemple, C++). Le tableau 3.1 adapté de leur article (et librement traduit) liste quelques langages généralement considérés comme des DSL.

DSL	**Domaine d'affaires**
BNF	Grammaire
Excel	Tableur
HTML	Pages Web
LaTeX	Typographie
Make	Moteur de production de logiciels
SQL	Requêtes de bases de données

Tableau 3.1 Exemples de langages spécifiques au domaine (23).

[2] http://fr.wikipedia.org/wiki/Forme_de_Backus-Naur

3.2.2 Inconvénients des DSL

Les langages spécifiques au domaine n'ont pas que des qualités. Ils doivent être appris des utilisateurs, problème que nous avions déjà vu avec OtoScript. Leur conception, mise en œuvre et maintenance sont difficiles, ce dont nous avons également fait l'expérience. Peu d'outils sont disponibles pour faciliter leur utilisation, ou, du moins, il sera nécessaire de les étendre en fonction des mots-clefs du DSL (coloration syntaxique, notamment).

Au niveau de la performance, comme le fait remarquer l'article éponyme de Wikipédia[3], les programmes développés dans des DSL utilisent habituellement moins efficacement le temps processeur, étant à un plus bas niveau d'abstraction et souvent interprétés, par rapport aux langage généralistes disposant la plupart du temps de compilateurs optimisants ou de machines virtuelles très performantes, comme la machine *HotSpot* dans le cas de Java.

3.2.3 Types de DSL

À l'intérieur même des DSL, nous pouvons distinguer deux approches différentes en terme de mise en œuvre : les DSL externes (*External DSL*) et les DSL internes (*Embedded DSL* ou *Internal DSL*) (13). Un DSL externe est un langage de programmation classique : du code source développé dans ce langage doit subir une analyse lexicale, syntaxique et sémantique avant d'être compilé ou interprété. OtoScript faisait partie de cette catégorie : les phases d'analyse syntaxique et sémantique étaient supportées par son traducteur externe, alors que l'interprétation était effectuée par le moteur d'Oto. À l'opposé, un DSL interne vient se greffer à un langage existant et exploite ses fonctionnalités, ce qui lui permet d'hériter de plusieurs des caractéristiques de son langage hôte qu'il n'a pas à redéfinir explicitement.

Dans les prochaines sections, nous examinerons les avantages et les inconvénients de chacune de ces approches.

3.2.4 DSL externes

Les principaux avantages d'un DSL externe sont les suivants :

- Dans un DSL externe, la syntaxe du langage pourra représenter tout concept qu'il sera

[3] `http://en.wikipedia.org/wiki/Domain-specific_language`

souhaitable d'utiliser avec le domaine ciblé, permettant d'exprimer pleinement ce dernier en utilisant les termes et symboles qui lui sont propres.

- Le langage n'étant pas lié à un langage hôte, il ne devrait pas souffrir de biais de mise en œuvre ou de compromis liés à l'environnement dans lequel il sera intégré, notamment au niveau de la syntaxe ou de la plateforme cible (30).

Par contre, les inconvénients des DSL externes sont les suivants :

- La mise en œuvre d'un DSL externe est complexe. Comme il s'agit littéralement de développer un nouveau langage de programmation à partir de zéro, sa grammaire devra être définie, des analyseurs lexicaux, syntaxiques et sémantiques devraient être mis au point avant de pouvoir finalement le compiler ou l'interpréter. Toutes ces étapes demandent, en général, un effort non négligeable, notamment si une attention particulière est accordée à l'optimisation du code généré liée à des contraintes de performances.
- Une fois le langage développé, il sera nécessaire de lui offrir des fonctionnalités maintes fois supportées par une multitude de langages (telles que l'écriture de fichiers), ce qui est coûteux en temps.

3.2.5 DSL internes

Les avantages d'un DSL interne sont les suivants :

- Comparé à un DSL externe, le DSL interne est a priori plus simple à mettre en œuvre, car les étapes d'analyses lexicale, syntaxique et sémantique ainsi que la compilation ou l'interprétation seront effectuées par l'environnement du langage hôte. Par exemple, un DSL interne basé sur Java sera compilé par le compilateur `javac`.
- Le DSL interne peut hériter des structures et services offerts par le langage hôte, diminuant d'autant la tâche du développeur du langage spécifique au domaine.
- Dans un DSL interne, il sera habituellement possible d'accéder aux bibliothèques de code du langage hôte, code que le développeur du langage spécifique au domaine n'aura pas à réécrire ou à adapter.

Par contre, leurs inconvénients sont les suivants :

- Pouvant être considéré comme une extension du langage hôte, le DSL interne n'est pas entièrement libre de prendre toutes ses décisions de mise en œuvre et doit respecter les conventions et contraintes de l'environnement qu'il étend. Par exemple, en Ruby il ne sera pas possible d'utiliser le tiret (-) dans les noms de classes, de méthodes, d'attributs et de variables. Un DSL interne basé sur Ruby devra se plier à cette contrainte. Pour cette raison, il se pourrait que le langage hôte ne permette pas l'intégration d'éléments lexicaux ou syntaxiques désirables dans le DSL, pour lesquels des compromis devront être trouvés.[4]
- Un DSL interne peut être difficile à développer et à maintenir, notamment lorsque la syntaxe ou les structures désirées ne s'intègrent pas facilement dans le langage hôte.
- L'exploitation de structures du langage hôte hors de leur contexte naturel par le DSL interne (par exemple, utiliser un appel de méthode pour effectuer un calcul) entraîne des messages d'erreurs parfois difficiles à comprendre, étant générés par le langage hôte qui ne connait pas le contexte du DSL interne, ce qui contribue à complexifier le débogage.

3.3 Ruby et les DSL internes

Ruby est un langage convenant particulièrement bien au développement de DSL internes. Ses aptitudes à la métaprogrammation et à l'introspection, son typage dynamique, sa souplesse syntaxique au niveau de l'appel des méthodes[5] ainsi que la présence d'un bibliothèque de classe relativement riche en font un environnement intéressant pour ceux-ci. Nous verrons ces capacités plus en détails à la section 3.3.1.

Au moment de rédiger ce mémoire, plusieurs projets intéressants étaient construits autour du principe d'un DSL interne basé sur Ruby. Nous mentionnons quelques uns de ces projets au tableau 3.2, qu'une recherche rapide sur le Web nous a permis d'identifier.

---

[4] Ce qui, nous en convenons, peut parfois être contradictoire avec le principle d'utilisation d'un DSL.

[5] Où les parenthèses sont optionnelles dans le cas d'un seul appel par ligne.

Nom du DSL interne	Nature et domaine d'affaires
Aquarium	Programmation à aspects pour Ruby
Peerant	DSL pour la téléphonie
Rails	Framework Web
Rake	Moteur de production de logiciels semblable à Make
SQL Generation DSL	Générateur de code SQL
XMLBuilder	Générateur de code XML

Tableau 3.2 Quelques exemples de DSL internes basés sur Ruby trouvés sur le Web.

3.3.1 Aspects de Ruby intéressants pour les DSL internes

Un certain nombre d'aspects de Ruby rend intéressant son utilisation dans le cadre d'un DSL interne. Dans cette section, nous considérerons les plus importants d'entre eux.

3.3.1.1 Classes ouvertes

Ruby est capable de modifier dynamiquement le contenu de ses classes. Des classes et des méthodes peuvent être définies dynamiquement et retirées de l'environnement Ruby au cours de l'exécution. Cela peut être fait de plusieurs façons, mais la plus intéressante consiste tout simplement à saisir le nom de la classe n'importe où dans le programme, comme s'il s'agissait d'une classe que nous venions de définir, puis à définir le nouveau code. Par exemple, il serait possible d'ajouter une méthode à la classe `Time` pour afficher le temps en chiffres romains sans même ajouter une autre classe qui hériterait de la première. Le code peut aussi être ajouté en évaluant une chaîne de caractères contenant le code d'une classe dans le contexte de cette classe.

3.3.1.2 Typage dynamique

Ruby est un langage à typage dynamique. Dans un tel langage, une variable ne reçoit pas de type statique à la création (annotation de type), son type dépend entièrement de la valeur qui lui est assignée (type dynamique). Ce type est alors associé à la valeur, et non à la variable elle-même, par opposition à un langage à typage statique utilisant l'inférence de types.

Un langage à typage dynamique convient mieux pour le développement de DSL internes, car, dans un tel cas, il sera vraisemblablement plus facile de cacher le langage hôte que si les types

devaient être déclarés explicitement (par annotation) ou s'il n'était pas possible de réutiliser une variable parce qu'un type lui a été attribué à l'instanciation (inférence de types).

3.3.1.3 Réflexivité

Comme certains autres langages, notamment Java, Ruby est un langage réflexif. Une classe peut informer un requérant de ses méthodes, de ses variables d'instance et de ses super-classes. Surtout, en Ruby, toutes ces méta-informations peuvent être manipulées et modifiées. Ainsi, il sera possible de changer dynamiquement leur comportement.

3.3.1.4 Symboles

En Ruby, un symbole est une chaîne de caractères invariable et respectant le principe d'unicité, c'est-à-dire que la liste des symboles utilisés est conservée en mémoire de sorte qu'un symbole ne sera défini qu'une seule fois à travers un programme Ruby. Les différentes occurences d'un symbole dans le programme Ruby font toutes référence à une unique chaîne en mémoire, ce qui permet d'exploiter plus efficacement l'espace disponible que des chaînes de caractères conventionnelles dont il doit exister autant d'instances en mémoire qu'il y a d'occurences dans le programme. Dans la syntaxe du langage, un symbole est identifiable à ses deux-points précédant les caractères, tel que *:Un_symbole*. Au niveau de la mise au point d'un DSL, la classe *Symbol* peut être utilisée pour définir des méthodes sur les symboles permettant de définir le comportement désiré.

3.3.1.5 Envoi de messages et *method_missing*

Dans un langage à objets, un objet peut être en mesure de répondre à des méthodes qui sont invoquées sur lui. Le nom véritable de cette propriété est l'*envoi de messages*. Lorsque nous invoquons une méthode sur un objet receveur, nous lui envoyons en réalité un message portant un nom (le nom de la méthode) et possiblement des paramètres. Si l'objet possède une méthode correspondant au nom du message qu'il a reçu (celle-ci pouvant avoir été définie directement dans la classe définissant l'objet, ou dans l'une de ses super-classes), il exécute cette méthode en lui passant les paramètres fournis dans le message. Du point de vue de la hiérarchie de classes, la méthode sera recherchée dans la classe réelle de l'objet receveur, puis, si elle n'y est pas trouvée, dans les classes supérieures. Si, au terme de ce parcours, aucune méthode de ce nom n'a été localisée, le moteur d'exécution du langage soulèvera une exception indiquant que

la méthode correspondant au message qu'il a reçu est manquante.

Lorsque le type du receveur est connu statiquement, il sera possible de prévenir certaines de ces exceptions, car la classe ayant servi à construire le receveur sera connue, ce qui permettra de connaître les méthodes que cette classe possédait à ce moment. Dans un langage à typage dynamique, toutefois, il ne sera pas possible de savoir à l'avance si un objet sera en mesure de répondre ou non à un message qui lui est envoyé, car le receveur sera défini sans type statique ; seul l'envoi lui-même permettra de déterminer si l'objet supporte ou non le message, d'autant plus qu'on peut, via réflexivité, définir dynamiquement des méthodes.

L'un des aspects les plus intéressants de Ruby est qu'il permet d'intercepter les envois de message auxquels un objet n'a pu répondre. Pour ce faire, il faudra définir la méthode `method_missing` dans la classe définissant cet objet. Lorsque cette méthode est définie, l'envoi de messages se fera comme à l'habitude, sauf en présence d'un appel à une méthode inconnue. Dans ce cas, au lieu de poursuivre la recherche à un niveau hiérarchique plus élevé (ou soulever une exception si le niveau le plus élevé était déjà atteint), la méthode `method_missing` sera exécutée.

Une telle fonctionnalité peut servir à employer les appels de méthodes à d'autres fins propres à un DSL, notamment pour saisir un identificateur sans les guillemets d'une chaîne de caractères ou les deux-points d'un symbole Ruby. Le langage Ruby étant à typage dynamique, ses mécanismes exécuteront ces appels comme s'il s'agissait de méthodes conventionnelles, se contentant d'exécuter `method_missing`, qui devra effectuer le traitement qu'il juge approprié. Un exemple concret que nous présentons à la section 3.4 fait un usage abondant de `method_missing` dans sa mise en œuvre.

3.3.1.6 Surcharge des opérateurs

À l'instar de certains langages, notamment C++, Ruby permet la surcharge de la majorité de ses opérateurs. Ceux-ci ne sont en réalité que des méthodes qu'il suffit de redéfinir. Ces opérateurs peuvent ensuite être utilisés dans la mise en œuvre de la syntaxe du DSL. Le langage n'impose pas de restrictions sur l'utilisation qui sera ensuite faite des opérateurs de telle sorte que l'opérateur `&`, normalement assigné au ET logique, pourra être redéfini pour donner, par exemple, l'adresse mémoire d'un objet.

3.3.1.7 Évaluation dynamique

Une autre caractéristique intéressante de Ruby est sa capacité à évaluer une chaîne de caractère quelconque comme étant du code Ruby valide dans le contexte courant, ou dans celui d'une classe ou d'une instance. Le langage permet également de passer un bloc de code d'une instance à une autre instance, pour l'évaluer dans la seconde.

3.3.1.8 Syntaxe tolérante et sucre syntaxique

La syntaxe de Ruby est caractérisée par sa grande tolérance au niveau de l'utilisation des parenthèses dans l'appel de méthodes et par la possibilité d'omettre une partie des éléments syntaxiques nécessaires à l'appel de méthodes et au niveau des paramètres de méthodes. Par exemple, les parenthèses sont optionnelles dans un appel de méthodes à condition que l'appariement des paramètres ne soit pas ambigu (c'est-à-dire s'il y a un seul appel sur la ligne de code). De cette façon, *methode param* est équivalent à *methode(param).* Utilisé comme paramètre de méthode, les accolades délimitant un dictionnaire peuvent être aussi omises.

En cas d'utilisation d'un opérateur, qui revient à invoquer une méthode, l'opérateur point (.) est optionnel. Par exemple, l'expression `'chaine1' + 'chaine2'` signifie en réalité `'chaine1'.+( 'chaine2' )`. Dans un tel cas, il serait tout à fait possible de surcharger la méthode `+` pour lui assigner le comportement désiré. Ces fonctionnalités contribuent à utiliser Ruby pour développer un DSL ayant un syntaxe allégée.

3.4 Exemple de mise en œuvre d'un DSL interne basé sur Ruby

Dans cette section, nous survolerons un exemple de DSL interne basé sur Ruby pour comprendre comment ses auteurs sont parvenus à prendre avantage des structures et conventions de Ruby.

3.4.1 Description

L'exemple de DSL que nous présenterons ici est *XML Builder*, disponible à l'adresse `http://www.codecommit.com/blog/ruby/xmlbuilder-a-ruby-dsl-case-study`. Cette application sert à générer du code XML au moyen d'une syntaxe plus légère que celle du XML lui-même (évitant de devoir ouvrir et fermer des balises) permettant à l'utilisateur de se concentrer davantage

sur le contenu que sur la forme.

Cet exemple présente une structure de balises XML[6] dont certaines possèdent des attributs, des balises imbriquées ou des valeurs. L'idée est de faciliter la création par l'humain de fichiers XML en en retirant la syntaxe lourde pour se concentrer sur les informations et la structure. Le DSL permet ensuite de générer le fichier XML correspondant.

3.4.2 Utilisation de *XML Builder*

Un fichier XML valide se compose d'une balise *XML* qui contient, elle, d'autres balises définies par le domaine auquel ce fichier s'applique. Celles-ci peuvent également contenir, récursivement, d'autres balises. Une balise est définie par un nom entre crochets. Une balise ouverte doit par la suite être fermée : `<balise>...</balise>`. Si la balise ne contient pas d'autres balises, elle peut être fermée par la même commande que son ouverture : `<balise />`. Une balise peut posséder des attributs. Ceux-ci sont définis par un nom suivi d'une valeur placés dans la balise d'ouverture. La figure figure 3.1 présente un exemple de script *XML Builder* avec, en bas, le code XML correspondant.

Pour simplifier l'écriture du fichier, *XML Builder* retire les éléments syntaxiques (crochets, signe d'égalité) du XML pour ne conserver que les informations qu'il contient. Au lieu de crochets, le nom des balises est saisi directement. Une balise placée directement sous le premier niveau (dans l'exemple, le `<project>`) est créée en mentionnant son nom à la suite de l'objet `xml`, comme `xml.project`. Les attributs sont spécifiés à l'aide de la syntaxe des *hashes* Ruby, avec le nom à la gauche et la valeur à la droite de l'opérateur =>. Pour insérer des balises à l'intérieur d'une autre balise, la balise de niveau supérieur reçoit un bloc Ruby, délimité par les mots-clefs `do...end`. Le nom de ces balises est écrit directement, sans avoir recours à l'objet `xml`, comme les différents `target` de la figure 3.1. À la fin de l'exemple, le fichier XML généré est affiché à l'écran.

---

[6] *Extended Markup Language*, un ensemble de règles permettant de transporter et de stocker textuellement des informations de manière structurée. La particularité de ce langage est que le stockage se fait grâce à un ensemble de balises génériques imbriquées dont la syntaxe est semblable à celle du HTML. Un document XML doit être bien formé, c'est à dire satisfaire les règles strictes conditionnant les fichiers XML, et valide, ce qui signifie que son contenu doit respecter le modèle imposé par son développeur. Les caractéristiques du XML le rendent facile à manipuler par la machine, d'où la présence d'un nombre significatif de bibliothèques de code destinés à utiliser de tels documents.

```
require 'xmlbuilder'

xml.project :name => 'ActiveObjects', :default => :build do
  dirname :property => 'activeobjects.dir', :file => '$ant.file.ActiveObjects'
  property :file => '$activeobjects.dir/build.properties'

  target :name => :init do
    mkdir :dir => '$activeobjects.dir/bin'
  end

  target :name => :build, :depends => :init do
    javac :srcdir => '$activeobjects.dir/src', :source => 1.5, :debug => true
  end
end

puts xml

-----------------------------------------------------------------------------------------------

<?xml version="1.0" encoding="UTF-8"?>

<project name="ActiveObjects" default="build">
  <dirname property="activeobjects.dir" file="$ant.file.ActiveObjects"/>
  <property file="$activeobjects.dir/build.properties"></property>
  <target name="init">
    <mkdir dir="$activeobjects.dir/bin" />
  </target>
  <target name="build" depends="init">
    <javac source="1.5" debug="yes" srcdir="$activeobjects.dir/src"/>
  </target>
</project>
```

Figure 3.1 Comparaison du script *XML Builder* (en haut) et du code XML (en bas).

3.4.3 Mise en œuvre

La mise en œuvre du DSL interne de *XML Builder* est relativement simple. Dans celle-ci, les noms de balises sont en réalité des appels de méthodes. Les attributs de la balise sont chacun la clef et la valeur d'un *hash*. Celui-ci est passé en paramètre à la méthode appelée en spécifiant le nom de la balise. Finalement, l'intérieur des balises, soit une valeur, soit d'autres balises, est supporté par un bloc passé en paramètre à la méthode.

Utilisé normalement, l'appel à une méthode inexistante (le nom de la balise) soulèverait une exception. Or, pour modifier ce comportement, la méthode `method_missing`, à propos de laquelle nous avons élaboré à la section 3.3.1.5, est définie. Nous la présentons à la figure 3.2.

D'abord, le nom de la méthode inexistante (reçu par la méthode sous la forme d'un symbole, nommé `sym`) est utilisé pour former le nom de la balise.

Ensuite, la présence d'un paramètre passé à la méthode est vérifiée. En présence du paramètre attendu, c'est à dire un seul objet de type *Hash*, les clefs et valeurs de ce paramètre (nommé `args` dans la méthode) sont utilisés pour former les attributs de la balise.

Par la suite, la présence d'un bloc passé à la méthode (paramètre `block`) est vérifiée. Si aucun bloc n'a été fourni, la balise est fermée. Sinon, le contenu du bloc est évalué à l'aide d'une nouvelle instance de la classe du *XML Builder*, dans l'instance de laquelle le bloc est exécuté. Cette évaluation dynamique, capacité dont nous avions traités à la section 3.3.1.7, se fait à l'aide de la méthode Ruby `instance_eval`. De cette manière, cette analyse se fait récursivement jusqu'au dernier niveau d'imbrication. En dernier lieu, la balise fermante est ajoutée au résultat.

```
def method_missing(sym, *args, &block)
  @last = "<#{sym.to_s}"

  if args.size > 0 and args[0].is_a? Hash  # never hurts to be sure
    args[0].each do |key, value|
      @last += " #{key.to_s}=#̈{value.to_s}"
    end
  end

  if block.nil?
    @last += "/>"   # if there's no children, just close the tag
  else
    @last += ">"

    builder = XMLBuilder.new
    builder.instance_eval block

    @last += builder.last

    @last += "</#{sym.to_s}>"
  end
end
```

Figure 3.2 Code de la méthode `method_missing` de *XML Builder* (section 3.4).

CHAPITRE IV

MISE EN ŒUVRE DU DSL INTERNE ET DES EXTENSIONS

À quoi ressemble le DSL Oto? Quels sont ses principes inhérents? Comment a-t-il été mis en œuvre? Quels mécanismes de Ruby utilise-t-il pour accomplir ses objectifs? Comment l'avons-nous intégré à Oto? Comment avons-nous adapté les modules aux changements apportés aux scripts? Quelles sont les difficultés auxquelles nous avons fait face?

Dans ce chapitre, nous discuterons de la mise en œuvre des modifications que nous avons apportées à Oto 2 pour améliorer sa flexibilité et ses performances au moyen d'un nouveau mécanisme de coordination des tâches de correction de type DSL interne basé sur Ruby, appelé DSL Oto. Cette mise en œuvre concerne d'abord le DSL lui-même, et ensuite les modules d'extension, qui furent adaptés de ceux d'Oto 1+, les rapports, dont la mise en œuvre *a priori* similaire aux modules diffère néanmoins à plusieurs niveaux, puis l'adaptation du reste d'Oto, particulièrement les commandes, aux changements apportés aux scripts. Il sera également question de l'adaptation des tests intégrés à Oto ainsi que de divers problèmes auxquels nous avons fait face et des solutions trouvées pour y remédier.

4.1 Notre solution

Pour mettre en œuvre le DSL Oto, nous avons choisi Ruby comme langage hôte. Ruby répondait à plusieurs des exigences que nous avions mentionnées à la figure 2.2 : c'est un langage à objets, qui fournit des structures de boucles et de blocs. Une telle solution nous permettait de nous concentrer sur l'appel des modules, des rapports et des commandes, ainsi que la manipulation des résultats retournés par ceux-ci. De plus, cette approche réduisait la courbe d'apprentissage d'Oto par un utilisateur connaissant déjà Ruby.

4.1.1 Présentation du DSL Oto

Dans cette section, nous examinerons un exemple de script de correction pour Oto 2. Après l'avoir présenté, nous discuterons de ses concepts.

4.1.1.1 Exemple de script de correction Oto 2

La figure 4.1 présente un script Oto coordonnant un scénario de correction typique. Celui-ci vise à corriger un programme Java, d'abord en le compilant, puis en utilisant JUnit pour effectuer un test unitaire sur la classe obtenue. Les résultats sont ensuite affichés à l'écran sous forme de rapport.

D'abord, le script définit deux variables de type chaîne de caractères (`fclient` et `ctest`). Ces variables représentent respectivement le nom du fichier Java de l'étudiant et le nom de la classe de test fournie par l'enseignant.

Ensuite, l'objet `groupe` représente l'ensemble du groupe à corriger, lequel est utilisé pour manipuler tout ce qui doit concerner le groupe. Dans ce cas, nous allons exécuter des modules individuels, il faut donc manipuler chacun des éléments du groupe un à la fois. La méthode `each` nous permet de le faire : elle nous offre chacun des TP (`tp`) et nous permet de lui passer un bloc de code contenant les instructions à exécuter pour chacun.

Pour chacun des TP, nous commençons par copier le contenu de l'évaluation dans le répertoire du TP. Cette étape nous permet ici de copier la classe de test `ClientTest.class` fournie par l'enseignant en vue de son utilisation avec JUnit. Nous utilisons ensuite le module `compiler_javac` sur le TP pour compiler le fichier Java fourni par l'étudiant. Ce module s'exécutant sur le TP, nous devons le fournir en paramètre. Nous passons également un paramètre au module pour indiquer le nom du fichier Java à compiler : `:fichiers` >> `fclient`. Le résultat de la compilation, c'est-à-dire l'information retournée par le module, est placée dans la variable `res_javac`.

Ensuite, si la compilation a réussi (en fonction de l'information retournée par le module de compilation), le test unitaire est lancé à l'aide du module `tester_junit`. Ce module individuel a lui aussi besoin du TP, mais également d'un paramètre lui disant le nom de la classe de test à exécuter. Cela fait, nous plaçons le résultat dans la variable `res_test`. Le module `tester_junit` fournissant l'information sur le nombre d'erreurs qui se sont produites, nous pouvons utiliser

ce nombre pour vérifier diverses conditions. Si des erreurs se sont produites, nous écrivons une chaîne expliquant que la compilation a réussi, mais que des erreurs se sont produites. Le détail des erreurs est ensuite conservé dans le TP sous le nom `Details` pour être affiché ultérieurement dans le rapport de correction, à la ligne `tp[:Details] = res_complet[:detail]` (ligne qui ne conserve que le résultat `:detail` de l'objet `res_complet` pour l'inclusion dans le rapport, évitant d'inclure des informations superflues dans ce dernier). Si aucune erreur ne s'est produite, un message de félicitations est conservé à la place. Si, par contre, la compilation n'a pas réussi, aucun test unitaire n'est lancé et le message d'erreur produit par le module `compiler_javac` est conservé dans les détails de la correction. Le message à fournir à l'étudiant est conservé ici sous le nom `Message` avec une majuscule pour en faciliter la lecture dans le rapport, mais ce n'est en rien obligatoire.

Une fois le travail sur chacun des TP terminé, nous produisons un rapport de correction complet. Ce rapport reçoit tous les résultats que nous voulons y inclure, dans ce cas le groupe. Ce rapport est une chaîne de caractère. L'instruction Ruby d'affichage de chaîne `puts` affiche ensuite le rapport sur la sortie standard.

4.1.1.2 Concepts de TP et de groupe

Dans notre DSL, le TP représente le travail d'une équipe d'étudiants. Le paramètre `tp` est un objet contenant toutes les informations nécessaires à sa manipulation par les modules et à la conservation des résultats obtenus. Le groupe, quant à lui, contient l'ensemble des TP soumis à Oto pour la correction en cours. C'est également un objet, auquel nous faisons appel pour plusieurs tâches, notamment l'exécution de la correction sur chacun des TP, l'appel de modules collectifs et la production de rapports.

L'objet `tp` sert également à conserver les résultats obtenus à l'exécution. Pour ce faire, il faut utiliser l'opérateur crochets (`[]`). Entre les crochets, nous identifions le nom que nous souhaitons donner au résultat, ce que nous pouvons faire avec les guillemets simples, doubles ou au moyen d'un symbole[1] (par exemple `tp[:Details]`). Une fois conservé, le résultat sera accessible en lecture et en écriture, pouvant être utilisé pour produire un rapport ou être accédé par un module de correction.

---

[1] En Ruby, les identificateurs, précédés par les deux-points sont des symboles.

```
fclient = 'Client.java'
ctest = 'ClientTest'

groupe.each {  |tp|

  rep_script.exporterVers tp.repertoire

  res_javac = compiler_javac( tp ) {
    :fichiers >> fclient
  }

  if res_javac.reussi? then
    res_test = tester_junit( tp ) {
      :classe >> ctest
    }
    if res_test[:nberreurs] > 0 then
      message = 'Le travail a ete compile, mais contient une ou plusieurs erreurs.'
      tp[:Details] = res_test[:detail]
    else
      message = 'Le travail ne contient aucune erreur. Felicitations!'
    end
  else
    # La compilation n'a pas fonctionne, le signaler a l'etudiant
    tp[:Details] = res_javac.message_erreur_echoue
    message = 'La compilation du travail a echoue.'
  end

  tp[:Message] = message
}

res_rapport = produire_rapport_complet( groupe )
puts res_rapport
```

Figure 4.1 Exemple de script de correction pour Oto 2.

4.1.1.3 Appel de modules

Dans Oto 2, nous distinguons deux types de modules de correction, les modules individuels et les modules collectifs. Un module doit obligatoirement recevoir l'objet correspondant à la cible de la correction : un module individuel doit recevoir l'objet `tp` et un module collectif doit recevoir l'objet `groupe`. Cet objet est placé immédiatement entre parenthèses après le nom du module, avant le passage du bloc des paramètres, identifié par les accolades, bloc qui n'est pas obligatoire si les paramètres du module sont tous optionnels.

Comme dans Oto 1+, les modules reçoivent des paramètres dont certains sont obligatoires et d'autres optionnels. Sous Oto 2, les paramètres sont passés dans des blocs. Dans le bloc, chaque ligne représente un paramètre différent. Sur chaque ligne, deux informations sont écrites. D'abord, le nom du paramètre est écrit à la gauche (ici, le paramètre se nomme `fichiers`) et l'objet passé est placé à la droite (ici, `fichier.java`). L'opérateur que nous utilisons pour assigner une valeur à un paramètre dans l'appel du module, l'opérateur d'assignation >>, est obligatoire. Nous avons choisi cet opérateur, dont la syntaxe est identique à l'opérateur de flux de C++, car il s'agissait, parmi les divers opérateurs dont la surcharge est possible en Ruby, de celui qui nous semblait le mieux représenter l'« attachement » d'un attribut à une valeur.

```
res_javac = compiler_javac( tp ) {
  :fichiers >> 'fichier.java'
}
```

L'objet retourné par le module permet d'en traiter les résultats, notamment en indiquant si l'objectif du module a été satisfait. Dans Oto 2, nous introduisons le principe d'objectif de module. Chaque module doit définir un objectif qui devra être satisfait pour que la méthode `reussi?` retourne vrai. Pour `compiler_javac`, par exemple, l'objectif sera que la compilation a réussi sans erreur, pour `tester_filtre`, aucune différence ne se serait produite entre les résultats attendus et obtenus, et ainsi de suite. Cet objet, de type `ContexteResultat` pour les modules individuels (ainsi que les commandes `bash`, comme nous le verrons sous peu) ou de type `ContexteResultatCollectif` pour les modules collectifs, conserve les résultats obtenus pour être, si nécessaire, traités par un ou plusieurs rapports.

4.1.1.4 Appel de rapports

Dans Oto 2, les rapports ressemblent à des modules de correction, mais leurs paramètres sont différents. Ils reçoivent le groupe et/ou un ou plusieurs résultats retournés par des modules collectifs. Le rapport peut également recevoir des paramètres dans un bloc comme un module. Il retourne une chaîne de caractères ou rien (en Ruby, *nil*).

Alors que le rapport était monolithique, non modulaire et s'affichait toujours dans les versions initiales d'Oto, Oto 2 augmente sa souplesse et ses caractéristiques. Le rapport n'est pas affiché par défaut ; pour ce faire, il faut utiliser une instruction de sortie comme `puts`.

4.1.1.5 Commandes Unix

Les scripts d'Oto 2 sont faits pour faciliter l'utilisation de commandes *bash*. Le DSL Oto permet d'utiliser les commandes de deux manières différentes. Dans la plupart des cas, il suffit de saisir la commande entre accents graves (en anglais, *backquotes*) pour qu'elle soit exécutée. Cela convient particulièrement si un message d'erreur éventuel doit être retourné immédiatement à l'utilisateur (par exemple, si un fichier devant être copié est introuvable), car cette manière d'exécuter les commandes affichera les sorties d'erreurs là où elles se produisent dans le script.

Par contre, l'utilisation des *backquotes* n'est pas toujours désirable. Par exemple, si la commande *make* est utilisée pour lancer des corrections, il sera plus intéressant d'intercepter les erreurs et de les afficher dans la section correspondant au TP en cours de correction dans le rapport final. Pour utiliser les commandes de cette manière, il ne sera pas nécessaire d'avoir recours à un module ou à des instructions particulières, il suffira de les saisir au moment du script où il faudra les exécuter. Dans le script suivant, Oto exécute la commande `ls -l` pour afficher le contenu détaillé du répertoire courant (par défaut, le répertoire à partir duquel Oto a été exécuté). Après l'exécution de la commande, le contenu des canaux de sortie standard et d'erreur sont placés dans un objet résultat, conservé dans l'exemple dans la variable `res_ls`. Le script affiche ensuite le contenu de la sortie standard au moyen de l'attribut `:STDOUT` de cet objet. Il aurait été possible de récupérer le contenu de la sortie d'erreur en utilisant l'attribut

`:STDERR` de la même manière.

```
groupe.each { |tp|
  `cp ~/le_fichier #{tp.repertoire}`
  res_ls = ls '-l'
  puts res_ls[:STDOUT]
}
```

4.1.1.6 Commentaires dans le code des scripts

Dans le script, comme en Ruby, tout ce qui suit le croisillon (#) sur une ligne est considéré comme un commentaire (au sens de langage de programmation, et non au sens de commentaire de rapport que nous verrons à la section 4.1.1.7) et sera ignoré. Les commentaires sur plusieurs lignes sont supportés, à l'instar de son langage hôte, avec les marqueurs `=begin` et `=end`.

4.1.1.7 Décoration du rapport

Dans les versions initiales d'Oto, une décoration de script était une chaîne de caractères placée entre guillemets doubles qui était ajoutée comme identificateur dans le rapport à la fin de la correction. Elle servait à clarifier et faciliter la lecture du rapport.

Toutefois, Oto 2 ayant modifié le fonctionnement des rapports, la décoration a été remplacée par les commentaires de résultats. Ceux-ci vont remplacer le nom du résultat dans le rapport pour rendre celui-ci plus explicite tout en évitant des lignes trop longues dans le script de correction. Examinons l'exemple suivant :

```
groupe.each { |tp|
  tp['Résultat obtenu à la compilation du fichier Test.java'] = compiler_javac( tp ) {
    :fichiers >> 'Test.java'
  }
  if tp['Résultat obtenu à la compilation du fichier Test.java'].reussi? then
...
```

Pour chacun des TP, le résultat de la compilation est conservé sous un nom significatif, mais dont l'usage alourdit le script. Le commentaire de résultat permet de conserver le résultat

sous un nom plus court, mais qui sera remplacé par le nom désiré dans le rapport, comme par exemple :

```
groupe.each { |tp|
  tp['res_javac'] = compiler_javac( tp ) {
    :fichiers >> 'Test.java'
  }
  if tp['res_javac'].reussi? then
...
end
groupe.commentaire_resultat(
  'res_javac',
  'Résultat obtenu à la compilation du fichier Test.java' )

...
```

Notons qu'il est également possible d'arriver au même résultat au moyen de n'importe quelle expression retournant une chaîne de caractères ou un symbole. Nous pouvons, par exemple, définir une variable contenant une chaîne et l'utiliser comme identificateur :

```
res_javac = 'Résultat obtenu à la compilation du fichier Test.java'
groupe.each do |tp|
  tp[res_javac] = compiler_javac( tp ) {
    :fichiers >> 'Test.java'
  }
  if tp[res_javac].reussi? then
  ...
end
...
```

4.1.1.8 Autres caractéristiques héritées de Ruby

Certains des aspects qui manquaient à OtoScript et que nous désirions rendre disponibles sont offerts directement par Ruby, notamment les boucles. De plus, ces nouveaux scripts étant objets, il est possible de créer des classes et des méthodes à l'intérieur des scripts pour rendre ceux-ci plus puissants.

Les scripts étant basés sur Ruby, celui-ci fournit sa bibliothèque de classes à l'utilisateur d'Oto, permettant aux scripts de lancer des `threads`, de communiquer par le réseau avec la bibliothèque `Net`, etc. Ces caractéristiques en font un langage nettement plus puissant qu'OtoScript sans que nous ayons eu à mettre en œuvre ces classes.

4.1.1.9 Aspects d'OtoScript non supportés dans Oto 2

Certaines des caractéristiques d'OtoScript n'ont pas été conservées ou ont vu leur rôle modifié dans Oto 2. Parmi celles-ci, mentionnons les variables globales et le concept de variables et d'appel de modules publics et privés.

D'abord, OtoScript offrait un concept de variables globales. Ces variables, déclarées dans le script, étaient ensuite visibles de l'intérieur des modules. Nous avons décidé de ne pas reproduire ce mécanisme dans le DSL Oto pour deux raisons. Premièrement, il était désormais possible d'utiliser directement des variables globales de Ruby dans les scripts, ce qui rendait peu intéressant la mise au point d'un mécanisme similaire. Deuxièmement, la présence de variables globales dans les scripts permettait aux modules de communiquer avec Oto autrement que par leurs paramètres, contribuant à violer le principe de moindre surprise.

Ensuite, les concepts de variables et d'appels de modules publics et privés n'étaient plus utiles dans le DSL Oto, les rapports ayant été découplés du contenu et de l'ordre d'exécution des scripts. Au lieu de déclarer une variable ou un appel privé, pour qu'il ne soit pas affiché dans le rapport, il suffit de ne pas le conserver dans le TP ou, dans le cas d'un appel de module collectif, de ne pas l'inclure dans les paramètre du rapport.

4.1.2 DSL Oto et les standards de Ruby

Fondamentalement, un script Oto est un programme Ruby qui est exécuté dans le contexte d'Oto. La ressemblance avec un programme Ruby standard est frappante : les scripts utilisent les structures d'Oto, la manipulation de collections se fait au moyen de blocs et de la méthode `each`. Les scripts possèdent toutes les capacités de programmes Ruby, pouvant déclarer et utiliser des classes et des méthodes, importer des fichiers et des classes, utiliser les aptitudes de Ruby à l'introspection et à la métaprogrammation, etc. Ils respectent les mêmes contraintes, notamment l'héritage simple de classes.

Les scripts Oto, étant exécutés dans le contexte d'Oto, disposent de l'interface de programmation[2] que nous avons développée pour la gestion des tâches de correction, ce qui augmente leurs capacités. De plus, au lieu de retourner une exception, l'appel à une méthode manquante[3] lancera l'exécution d'une commande `bash` éponyme. Nous avons également défini la méthode >> de la classe `Symbol` pour permettre son utilisation pour l'assignation de valeurs dans les paramètres passés aux rapports et aux modules.

L'utilisation d'un DSL interne basé sur Ruby nous a contraints à des compromis au niveau de la syntaxe. Nos scripts doivent notamment respecter les conventions de nommage de Ruby, qui interdisent entre autres l'utilisation du tiret (-) dans les identificateurs. Ils doivent également respecter l'utilisation des *sigils*, c'est-à-dire un ou plusieurs caractères précédant l'identificateur, pour la portée des variables, telle que définie en Ruby : une variable préfixée par un $ est globale, une variable d'instance sera préfixée par un seul arrobase (@) alors qu'une variable de classe sera préfixée par deux arrobases. Les variables locales ne sont pas préfixées. Il nous a également été impossible d'avoir recours à l'opérateur `=>` pour l'assignation de valeurs dans les paramètres passés aux modules et aux rapports, car celui-ci était réservé pour les *hashes*. Notre solution (>>) semble un compromis acceptable.

4.2 Mise en œuvre du DSL

Dans cette section, nous discuterons de la mise en œuvre du DSL dont la présentation a été faite à la section précédente.

4.2.1 Principes de mise en œuvre

4.2.1.1 Niveau d'inférence entre le DSL et le Ruby sous-jacent

À un haut niveau, le script Oto peut être vu comme un programme Ruby valide (le script) qui est exécuté à l'intérieur de l'instance d'un autre programme Ruby (Oto). Si une telle approche est avantageuse au niveau des performances (en évitant de lancer le script dans une autre instance de Ruby, les surcoûts qui seraient associés à l'exécution de cette autre instance et à la communication avec l'instance d'Oto sont épargnés) ainsi qu'au niveau de la réutilisa-

[2]En anglais, *application programming interface* (API).

[3]C'est à dire un message auquel aucun objet n'a répondu.

tion du code (les scripts pouvant accéder aux services offerts par Oto), elle fragilise toutefois l'application. En effet, comment une erreur survenant à l'exécution est-elle gérée par Oto? Quelles en sont les conséquences sur l'exécution de l'outil?

L'exécution du script Oto se fait en lisant d'abord le fichier du script comme un fichier texte, en chargeant son contenu dans une variable, puis en en lançant l'exécution au moyen de la méthode `instance_eval`. Celle-ci se fait à l'intérieur d'un bloc de code `begin..rescue` qui, en Ruby, permet d'intercepter les erreurs et exceptions survenant à l'exécution : le script pourrait être mal écrit et ne pas être un programme Ruby valide, ou une exception telle qu'une division par zéro pourrait survenir à l'exécution. Dans un tel cas, Oto intercepte le problème, le signale à l'utilisateur et termine sa propre exécution proprement, en nettoyant ses traces. Le message qui sera affiché est l'exception Ruby telle quelle, sans analyse ou simplification. Comme nous l'avons vu à la section 3.2.5, ceux-ci seront parfois peu significatifs et difficiles à interpréter. Il s'agit là d'un des inconvénients des DSL internes.

Nous présentons à la figure 4.2 le message qui sera retourné à l'utilisateur lors d'un problème survenant à l'exécution qui sera, dans cet exemple, provoqué par une division par zéro.

```
EXCEPTION_RUBY_DANS_LE_SCRIPT( exception_lancee = (eval):1:in '/': divided by 0 )
```

Figure 4.2 Exemple d'exception survenant à l'exécution (division par zéro).

4.2.1.2 Noms réels et alias

Comme nous le verrons à la section 4.3.3, l'utilisation de noms plus significatifs peut faciliter l'utilisation des modules en présentant leur rôle de manière plus claire. En même temps, un nom comme `produire_rapport_complet`, bien qu'explicite, demeurait lourd à écrire. Dans le même ordre d'idée, bien qu'il aurait été souhaitable de conserver le nom du répertoire d'un TP comme étant `rep_tp` pour demeurer fidèles aux versions initiales d'Oto, un mot comme `repertoire`[4] était plus clair. Dans les deux cas, pouvoir utiliser l'un ou l'autre nom représentait une solution intéressante.

La mise en œuvre de la solution dépend de la nature de l'appel auquel associer un alias. Pour le nom d'une méthode, il était possible d'utiliser le mot-clef Ruby `alias` pour que la

[4]Utilisé ici sans accent, pour correspondre à son utilisation dans un script Oto.

méthode puisse répondre à un envoi de message destiné à l'un ou l'autre nom. Au niveau des modules et des rapports, nous avons mis au point un mécanisme d'alias qui vient substituer le nom réel du module en cas d'utilisation d'un alias au moyen d'un *hash* Ruby. Le *hash* est initialisé dans le fichier de l'environnement d'exécution, comme dans l'exemple suivant, où l'alias est utilisé comme clef (à gauche) et le nom réel comme valeur associée (à droite) :

```
@alias_modules_rapports = {
  :produire_rapport_complet => :rapport_complet,
}
```

4.2.2 Utilisation des mécanismes de Ruby par le DSL

Les scripts Oto 2 sont basés sur l'utilisation sous-jacente de Ruby (appels de méthodes, traitement des paramètres de méthodes, passage de blocs, etc.) pour coordonner l'utilisation des modules, rapports et commandes `bash`, ainsi que pour assurer le traitement des résultats. Pour mieux illustrer comment Ruby « comprend » un script, nous présentons à la figure 4.3 un exemple découpant le script Oto en structures reconnues par Ruby.

Dans l'exemple, pour chacun des TP du groupe, nous effectuons certaines tâches de correction. D'abord, nous avons recours à la commande `cp` avec l'appel système de Ruby (les *backquotes*) pour copier un fichier vers le répertoire courant. La méthode `each` de l'objet groupe positionnant le répertoire courant (*pwd*) dans le répertoire temporaire associé au TP actuellement corrigé, il suffira de spécifier le répertoire courant comme destination de la copie.

4.2.2.1 Traitement des appels de méthodes

Nous avons défini `method_missing` pour traiter les instructions nécessaires à la manipulation des modules, rapports et commandes n'utilisant pas les *backquotes*. Dans l'exemple, `method_missing` recevra `compiler_javac` et le cherchera parmi les noms de modules et de rapports. Il le trouvera et chargera le module correspondant. En plus de recevoir entre parenthèses le paramètre `tp`, qui est l'objet du TP en cours de correction obtenu de la méthode `each` (au début du bloc), cet appel de méthode reçoit également un bloc contenant les paramètres destinés au module invoqué.

4.2.2.2 Blocs

En Ruby, un bloc permet entre autres de passer une série d'instructions en paramètres à une méthode. Dans le DSL Oto, le bloc sert à passer des paramètres aux modules et aux rapports. Le bloc est délimité par les instructions `do...end` ou par les accolades `{...}`.

4.2.2.3 Surcharge des opérateurs

Dans notre exemple, nous souhaitons que le résultat de l'appel du module `compiler_javac` soit conservé en vue d'être éventuellement inclus dans un rapport. Pour ce faire, nous plaçons l'objet retourné par le module dans le `tp` courant. Ruby permet de le faire de manière élégante en surchargeant l'opérateur crochets-assignation (`[]=`). Un symbole ou un `String` placé entre les crochets sert d'identificateur à la valeur qui y est assignée. La consultation des résultats, comme nous la faisons à la ligne suivante de l'exemple (pour vérifier si la compilation a réussi) se fait en surchargeant l'opérateur crochets (`[]`). La méthode `[]` retourne le résultat associé à l'identificateur qui lui est passé, ou soulève une exception si l'identificateur est inconnu.

4.2.3 Interface de programmation des scripts

La classe `EnvironnementExecution`, qui englobe l'exécution des scripts, leur offre un certain nombre de services pour qu'ils puissent accomplir leurs tâches. Ceux-ci sont mis en œuvre comme des attributs et des méthodes codées en dur dans la classe et relatives au contexte de correction.[5] Pour éviter de surcharger ce chapitre, nous les présentons à l'appendice E.

4.2.4 Organisation des classes

Les classes Ruby mettant en œuvre le DSL peuvent être regroupées en deux catégories. D'abord, plusieurs classes forment le DSL lui-même et tous les mécanismes nécessaires à l'exécution des scripts : charger le script, encadrer l'exécution en traitant les appels de modules, de rapports et de commandes `bash`, traiter les paramètres passés aux modules ainsi que fusionner les résultats en vue de la production des rapports. Le répertoire `DSL` regroupe ainsi toute la partie « mécanique » de l'exécution des scripts, et vient essentiellement se substituer à l'ancien

[5] Il va de soi que cette liste de services s'ajoute aux variables globales définies dans Oto et à celles mises en place par Ruby, comme `ENV` pour les variables d'environnement.

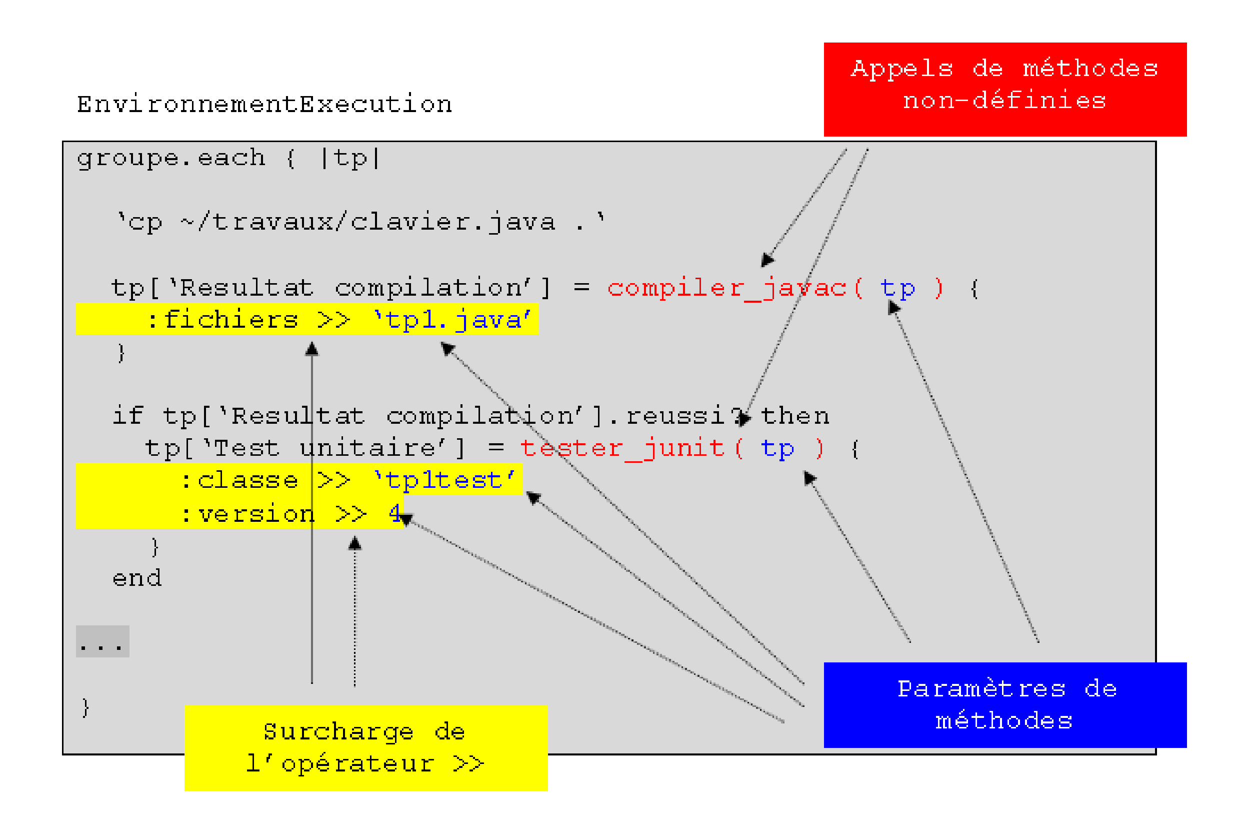


Figure 4.3 Exemple d'utilisation des structures de Ruby par un script Oto.

moteur des versions initiales d'Oto. L'autre catégorie, le contexte de correction, regroupe toutes les classes servant à représenter le groupe de TP corrigés et à traiter les résultats obtenus, tant en lecture qu'en écriture. Ce contexte vient remplacer les classes de l'ancien contexte d'exécution qui comprenait aussi une partie de la production des rapports.[6] Ces classes sont placées dans le répertoire `contexte`.

4.2.5 Concepts de métaprogrammation utilisés

Pour réaliser notre DSL, nous avons eu recours à plusieurs des capacités de métaprogrammation de Ruby. Les plus importants de ces concepts sont regroupés au tableau 4.1.

Concept	**Exemple d'utilisation**
Ajout dynamique de méthodes	Pour éviter les collisions, injection de méthodes pour la récupération d'informations dans les blocs de paramètres et modification de la classe `Symbol`.
Ajout dynamique de variables d'instance	Résultats de modules collectifs.
Consultation de la liste des méthodes et des variables	Toutes les classes du contexte de correction et `EnvironnementExecution`.
`instance_eval`	Pour lancer l'exécution du script dans le contexte d'`EnvironnementExecution`.
`method_missing`	Pour les appels de modules, de rapports et de commandes.
Retrait dynamique de méthodes	Le `initialize` de `EnvironnementExecution`, après utilisation.

Tableau 4.1 Concepts de métaprogrammation utilisés dans le DSL Oto.

4.2.6 Modifications comportementales

Dans cette section, nous examinerons certains changements apportés au comportement d'Oto à certaines étapes de la correction, et les problèmes à la base de ces changements.

[6] À l'exception des rapports obtenus à l'utilisation de `corriger_groupe`, où la commande effectuait une partie du travail de formatage du rapport.

4.2.6.1 Retrait de l'indicateur de dossier courant

Les nouvelles capacités de correction ayant retiré la nécessité d'inclure un indicateur du TP en cours de correction dans le groupe, ce qui était fait par `corriger_groupe`, nous avons pu la supprimer. Cet indicateur était utile pour les modules visant le groupe entier, comme la détection du plagiat. Étant donné que la manipulation de chacun des TP était désormais explicite, et que les modules de groupe recevaient un objet leur permettant de traiter le groupe entier, cette solution temporaire n'avait plus de raison d'être.

Un tel changement accélérera l'exécution tant de la commande `corriger_groupe` elle-même (en évitant une écriture disque par TP corrigé) que du module `detecter_plagiat`, qui évitera une lecture disque intrinsèquement lente et coûteuse.

4.2.6.2 Copie des fichiers du script dans les répertoires

Contrairement à ce qui se faisait dans les versions précédentes, les fichiers inclus dans l'activation de l'évaluation en sus du script ne sont pas copiés dans les répertoires des TP au début de la correction. La copie doit désormais être faite de manière explicite.

Nous avons effectué ce changement pour trois raisons. D'abord, en n'ajoutant rien au contenu des répertoires des TP, il sera plus facile de prendre acte de ce contenu ou de le modifier. Ensuite, en procédant ainsi, l'exécution du script est moins dépendante de la liste de fichiers inclus à l'activation de l'évaluation au niveau des fichiers requis à l'exécution, ceux-ci pouvant provenir d'ailleurs dans l'espace disque du correcteur. Finalement, la copie implicite semblait violer le principe de moindre surprise, notamment lorsqu'un fichier dans un TP était de même nom que celui inclus à l'activation de l'évaluation.[7]

4.2.6.3 Passage de paramètres au DSL

Les versions initiales d'Oto ne permettaient pas de passer de paramètres au script depuis la ligne de commande. Or, nous avons réalisé qu'une telle caractéristique rendait les scripts plus souples. Par exemple, pour le cours Programmation Parallèle (INF5170) de mêmes cas de tests

---

[7]Étant donné que dans ce cas, le fichier du TP ne serait pas écrasé. Une telle situation peut poser problème si un enseignant tentait d'écraser la version remise par un étudiant d'un fichier qu'il avait lui-même fourni, par exemple une interface Java.

devaient être exécutés avec une quantité variable de processeurs. Un paramètre à fournir au lancement de la correction serait, dans un tel cas, la liste du nombre de processeurs à utiliser. Dans le script Oto, les paramètres sont disponibles par un *hash*, `parametres`, où le nom du paramètre sera la clef.

4.2.6.4 Manipulation de variables d'environnement

Pour reprendre l'exemple précédent, la définition du nombre de processeurs accessibles par un programme parallèle MPD (3) se fait en définissant une variable d'environnement. Sous les versions initiales d'Oto, il fallait utiliser des scripts `bash`, ce qui alourdissait l'utilisation d'Oto. Dans les scripts du DSL Oto, la manipulation de variables d'environnement se fait comme dans un programme Ruby : le *hash* global `ENV` associe le nom des variables et leur contenu (qui doit être de classe *String*). Cette fonctionnalité réduit le recours à des fichiers tiers.

4.3 Modules

Dans le DSL Oto, les modules d'extension conservent un rôle important dans la satisfaction des tâches de correction. Ils disposent désormais de plus grandes capacités, tant dans les paramètres qu'ils reçoivent, désormais des objets, que dans leur potentiel au niveau de la correction. Dans cette section, nous examinerons la mise en œuvre des modules dans Oto 2, particulièrement au niveau des changements apportés par rapport aux versions initiales de l'outil.

4.3.1 Modifications générales apportées aux modules

Comme nous l'avons vu à la section 4.1.1.3, nous avons divisé les modules en deux catégories, individuels et collectifs. Ceux-ci ne diffèrent qu'au niveau des données qui leur sont fournies et des résultats attendus : alors que le module individuel reçoit un seul TP et retournera un résultat individuel, le module collectif recevra le groupe entier et retournera un résultat collectif.

La structure des paramètres du module ou du rapport définit, à partir d'Oto 2, le type du paramètre. Une telle modification était nécessaire en raison du passage d'objets comme paramètres. Cette validation se fait à l'aide de la méthode `is_a?` qui permet de vérifier l'appartenance d'un objet à une classe ou à une de ses sous-classes.

4.3.2 Ajout et suppression de modules

À partir d'Oto 2, les modules `cmd_bash` et `junit4`[8] disparaissent. Le premier est remplacé par l'appel de commandes directement dans les scripts, alors que le second est fusionné avec le module JUnit régulier, le choix de la version utilisée étant fait par un paramètre optionnel. Nous avons également ajouté le module collectif `produire_statistiques` permettant de compiler des statistiques sur un résultat présent dans les TP.

4.3.3 Renommage de certains modules

Nous avons profité des modifications apportées à Oto pour renommer les modules qui ont étés conservés et adaptés des versions initiales. Un tel renommage avait deux objectifs. D'abord, expliciter le nom du module rendait plus évidente la tâche que celui-ci cherchait à accomplir (par exemple, `detecter_plagiat` à la place de `plagiat`). Ensuite, sans renommage des modules, l'appel de commandes Unix directement dans le script aurait provoqué des conflits de noms lorsque le nom de la commande correspondait à celui d'un module (par exemple, appeler directement le compilateur `javac` au moyen de la ligne de commande alors qu'il existe un module de même nom).

4.3.4 Modifications apportées aux modules

L'interface des modules a été modifiée sous Oto 2. En plus d'une nouvelle variable à déclaration obligatoire, nous avons ajouté au module la connaissance du TP ou du groupe sur lequel il s'appliquera. De plus, bien que le nom n'ait pas changé, le « contexte » reçu par le module ne concerne plus le seul TP en cours de correction, mais bien le contexte de correction complet, avec des informations sur le script et le groupe entier. Une telle capacité permet la correction croisée basée sur l'ensemble des travaux[9] et, du fait, augmente les capacités des modules.

Hormis l'ajout des nouveaux champs à déclaration obligatoire et l'ajout des types de

---

[8] Ce module était une copie du module `junit` qui utilisait une version plus récente du cadre de tests. Les modifications apportées à son interface et à ses résultats ne permettaient pas d'utiliser le module initial sans perdre la compatibilité avec l'ancienne version.

[9] Ce que les versions initiales d'Oto ne permettaient pas de faire (17).

paramètres, les modifications apportées aux modules individuels se sont, en général, limitées à retirer les conversions de type pour les paramètres numériques (qui ne sont plus passées comme chaînes de caractères) et remplacer les structures utilisées pour retourner les résultats par des objets destinés à cette fin. Il a également été nécessaire de définir, pour chacun, l'objectif de module, qui s'est limité à ne pas trouver d'erreur à la compilation pour les modules `compiler_gcc` et `compiler_javac` et à n'obtenir aucune erreur à l'exécution pour `tester_filtre` et `tester_junit`.

Par contre, le portage du seul module « collectif » déjà présent dans Oto 1+ fut plus difficile. Contrairement aux modifications apportées aux modules individuels, qui n'ont demandé que des changements quasi cosmétiques, il a été nécessaire de revoir en profondeur le module `detecter_plagiat` pour lui permettre de prendre avantage des capacités du DSL Oto. Il faut dire que ce module avait été développé pour contourner les défauts des versions initiales d'Oto, notamment par l'emploi d'un fichier indiquant le travail en cours de correction. De plus, il manipulait les fichiers originaux et non les copies temporaires faites par Oto pour éviter les faux positifs provoqués par des fichiers éventuellement inclus avec l'évaluation et copiés dans les répertoires.

Le module `detecter_plagiat` commence par valider ses paramètres. Par la suite, et il s'agit là d'une différence fondamentale avec Oto 1+, il manipule l'objet `groupe` et, pour chacun des TP, le compare avec les autres du groupe, alors que le module original ne traitait qu'un seul TP à la fois. Si des modifications doivent être faites aux fichiers avant la comparaison (par exemple, pour y retirer un squelette, tel que vu à la section 1.4.2), elles peuvent être faites efficacement et sans laisser de traces.[10]

4.3.5 Mise en œuvre des résultats de modules

Bien que d'apparence simple, la mise en œuvre des résultats de module n'en demeure pas moins non triviale. Dans cette section, nous traiterons de certains détails de mise en œuvre dont nous avons du tenir compte lors du développement d'Oto 2.

---

[10] Ce qui se produisait sous Oto 1+ lorsque les fichiers intermédiaires étaient conservés à la fin de l'exécution pour hausser les performances.

4.3.5.1 Préservation de l'ordre d'insertion

Pour faciliter la lecture des rapports, nous avons cru nécessaire que l'ordre d'acquisition des informations destinées à y figurer soit conservé. À cet effet, nous avons recours à une version modifiée des *hashes* Ruby conventionnels grâce à une sous-classe dont la méthode `each` retourne les éléments dans l'ordre d'insertion.[11]

4.3.5.2 Métaprogrammation et résultats collectifs

Si la mise en œuvre des résultats de modules individuels est relativement simple (surcharge d'opérateurs et utilisation d'un *hash* ordonné), celle des modules collectifs, devant conserver l'information pour un nombre indéterminé de résultats sur un groupe entier, s'avère plus complexe. Notre solution est venue de la métaprogrammation : si nous pouvons contrôler dynamiquement les attributs d'un objet, pourquoi ne créerions-nous pas à l'exécution des attributs correspondants aux résultats obtenus?

À la création d'un objet de résultat collectif, seuls certains attributs prédéterminés sont créés. Une méthode permet de différencier la liste des attributs réels de l'objet (par métaprogrammation) et d'en retirer les éléments prédéterminés, ne laissant que ceux que nous aurons éventuellement ajoutés. Une autre méthode permet d'accéder en lecture et en écriture à un attribut dont on spécifie le nom. Ces méthodes sont manipulées lors de la conservation du résultat d'un TP dans le groupe en incluant le nom de l'identificateur et le résultat. D'abord, si l'attribut correspondant à l'identificateur n'existe pas, il est créé. Ensuite, nous accédons à l'attribut au moyen de la méthode Ruby `instance_variable_get`, qui retourne le *hash* associé au résultat, où nous plaçons la valeur en fonction du TP concerné. Le code de la méthode d'ajout d'un résultat est présenté ci-bas.

```
def ajouter_resultat( nom_resultat, nom_repertoire, valeur )
  var_resultat = "@#nom_resultat"
  self.instance_variable_set( var_resultat, HashFactory.hash_ordonne ) \
    unless self.instance_variables.include?( var_resultat )
  self.instance_variable_get( var_resultat )[nom_repertoire] = valeur
```

[11] Par défaut, les *hashes* de Ruby 1.9 et ultérieures conservent l'ordre d'insertion. Toutefois, au moment d'écrire ces lignes, Oto demeure destiné à Ruby 1.8.6, notamment parce que les versions 1.9 et ultérieures modifiaient certains aspects fondamentaux du langage et demeuraient expérimentales.

```
end
```

4.3.5.3 Résultats collectifs : quoi inclure dans le rapport?

Au niveau des résultats de modules collectifs, le DSL Oto permet de restreindre les résultats à un ou plusieurs ensembles qui seront affichés dans le rapport. Ces trois ensembles permettent l'inclusion de différents résultats. Par défaut, un résultat collectif fournira deux ensembles au rapport. D'abord, un résultat collectif, regroupant les TP ayant réussi puis ceux ayant échoué l'objectif du module, afin de les afficher ensemble. Ensuite, pour chacun des TP, un résultat semblable aux résultats individuels qui détaille le résultat propre au TP dans la correction de groupe. Il est également possible d'afficher une autre information, soit une simple chaîne de caractères d'informations détaillées sur la correction.[12] Le choix de fournir un ou l'autre, deux ou les trois ensembles est défini dans le module collectif, en fonction de son rôle et de ses besoins.

4.4 Rapports

Par rapport aux versions initiales, les rapports du DSL Oto sont plus souples d'utilisation et plus puissants. En étant modulaires et en ne s'affichant plus systématiquement sur la sortie standard à la fin de l'exécution du script, ils offrent plus de flexibilité à l'utilisateur (au prix d'une opération supplémentaire, l'appel du rapport dans le script). Dans cette section, nous traiterons de la mise en œuvre des rapports, notamment au niveau de compromis nés du choix d'un DSL interne et des limites de manipulations de blocs Ruby.

4.4.1 Mise en œuvre des rapports des versions initiales

Dans les versions initiales, le contenu des rapports était fortement couplé à la présentation des éléments dans les scripts, ainsi qu'au moteur d'exécution et au contexte de correction et à la commande `corriger_groupe` au niveau de la mise en œuvre. Un tel éparpillement rendait difficile la compréhension de leur fonctionnement et complexifiait l'entretien du code.

---

[12] Par exemple, `produire_statistiques` servant à compiler les informations sur un résultat dans un groupe, il utilise seulement le détail pour afficher la moyenne, l'écart type et autres informations sur la correction, n'ayant rien à préciser sur chacun des TP.

4.4.2 Les rapports sous Oto 2

Sous Oto 2, les rapports ne sont plus produits par le moteur d'exécution des scripts. Ils deviennent des entités indépendantes, semblables aux modules, recevant les résultats obtenus à la correction ainsi que, possiblement, des paramètres, et retournant soit une chaîne de caractères soit simplement rien. La manipulation de cette chaîne est laissée à la discrétion de l'utilisateur.

4.4.2.1 Différences du mécanisme d'extension

Le fonctionnement des rapports est semblable à celui des modules, sauf au niveau des valeurs retournées, qui ne sont pas déclarées et se limitent à un objet de type `String` ou `nil` ainsi qu'aux paramètres reçus, qui doivent être placés dans un objet de type `Array` si plus d'un résultat doit être pris en compte dans le rapport. Le mécanisme d'alias des noms de modules fonctionne aussi pour les rapports (par exemple, les appels `produire_rapport_complet` et `rapport_complet` sont équivalents).

4.4.2.2 Fusion des résultats

Si plus d'un résultat doit être pris en compte pour la production du rapport, une fusion des résultats vers une forme intermédaire est faite avant l'appel au « module-rapport ». La fusion, supportée par la mise en œuvre du DSL (`fusionner_resultats.rb`), vise également à résoudre certains problèmes de noms, dont nous traiterons à la section 4.4.2.3. Les résultats sont regroupés par identificateur, et ensuite par TP, en tenant compte des commentaires d'identificateurs, découplant les rapports de la décoration des scripts et de la résolution de problèmes de noms.

4.4.2.3 Rapports et résultats de modules

Certains problèmes de noms surviennent avec les modules collectifs. Pour les modules individuels, conservés avec le groupe, et pour lesquels nous fournissons l'objet groupe pour les inclure dans le rapport, la solution est simple : le nom de l'identificateur utilisé pour le conserver sera repris dans le rapport, sauf si un nom de commentaire est présent, lequel viendra remplacer le nom de l'identificateur.

Toutefois, les résultats des modules collectifs étant sous forme d'objets, nous avons dû faire face à un problème : comment devions-nous les nommer? Malgré ses aptitudes à la

métaprogrammation, il n'était pas possible en Ruby de récupérer le nom donné à la variable recevant l'objet résultat. Face à cette situation, nous avons décidé, dans ce cas, d'employer le nom déclaré dans le module de correction comme identificateur dans le rapport. Il sera également possible de modifier ce nom grâce à un commentaire spécifié directement dans le résultat. Malgré cette solution, un conflit de nom éventuel demeurait si un module collectif était invoqué deux fois ou plus, le nom du module n'étant plus suffisamment discriminant. Dans ce cas, un marqueur numérique sera ajouté au nom du module et incrémenté autant de fois que le module sera invoqué, par exemple `Détection du plagiat` pour le premier appel, `Détection du plagiat-0` pour le second et ainsi de suite.

4.4.2.4 Rapport complet

Au moment d'écrire ces lignes, Oto 2 disposait d'un seul module-rapport. Celui-ci, nommé `rapport_complet`, produit un rapport contenant toutes les informations disponibles au sujet de la correction. Il présente d'abord un en-tête contenant, entre autres, le nom de l'évaluation (ou « temporaire » si la correction a été lancée par un fichier *.oto*, comme nous le verrons à la section 4.5.1), l'heure et la date de l'exécution et le contenu du script utilisé. Par la suite, dans l'ordre, les résultats de modules collectifs sont affichés, suivis de chacun des TP (pour chacun, les résultats individuels sont affichés dans l'ordre d'insertion) ainsi que les commandes utilisées hors groupe.

4.5 Intégration à Oto

Dans cette section, nous traiterons de certaines modifications que nous avons apportées à Oto pour y intégrer le DSL Oto ainsi qu'alléger l'utilisation de l'outil.

4.5.1 Exécution directe par fichier *.oto*

L'utilisation obligatoire des évaluations et des commandes de correction de TP et de groupe représentait une contrainte à l'utilisation d'Oto, car elle était lourde, exigeant l'activation d'une évaluation, son utilisation dans une correction et sa désactivation explicite. Dans le cas d'une erreur dans l'évaluation, il fallait la désactiver, corriger l'erreur dans le script et activer à nouveau l'évaluation, processus lourd et décourageant. Pour alléger l'utilisation de l'outil, nous avons ajouté la capacité de lancer Oto directement avec un fichier de script (*.oto*), les fichiers

nécessaires à l'évaluation ainsi que les TP à corriger en une seule commande. Dans ce cas, les deux seuls éléments obligatoires sont le script, qui doit apparaître en premier après l'appel à Oto sur la ligne de commande, et tous les TP à corriger. Oto activera une évaluation temporaire, nommée aléatoirement, et la désactivera à la fin de la correction ou si le script échoue.

4.5.2 Modification des commandes

Le passage au DSL Oto a également été l'occasion de revisiter certaines des commandes d'Oto, notamment pour les adapter à la nouvelle réalité, ou pour retirer certains défauts liés à la mise en œuvre des versions initiales.

4.5.2.1 Lancement des évaluations

Après avoir développé le DSL Oto, nous avons dû modifier les commandes capables de lancer l'exécution d'un script, y compris `rendre_tp` qui peut avoir besoin d'exécuter des scripts de remise de TP. Ces modifications n'ont suscité qu'une seule difficulté significative : Oto ne permettait pas de créer les dossiers temporaires de tout un groupe à la fois avant de corriger et ensuite supprimer ces dossiers. Il ne pouvait le faire que sur un seul TP à la fois. Pour corriger ce problème, nous ne nettoyons pas le dossier temporaire immédiatement après sa création, mais attendons à la fin de la correction.

4.5.2.2 Restrictions de boîte et remise de TP

À la remise d'un TP, Oto peut lancer une évaluation qui devra réussir pour que la remise soit acceptée. Cette évaluation est spécifiée à la création de la boîte de remise. À l'origine purement destinée à des vérifications telles qu'au moyen de JUnit, cette fonctionnalité a, par la suite, été exploitée par le professeur Tremblay lorsqu'il a souhaité ajouter la possibilité d'imposer, encore à la création de la boîte, une date limite de remise ou des listes précises de fichiers à remettre. Si l'idée était bonne, sa mise en œuvre reposant sur la génération de code OtoScript par des scripts *C-Shell* souffrait de deux défauts. D'abord, il n'était pas possible d'utiliser ces ajouts concurremment à un script de vérification, et ce uniquement pour des raisons de mise en œuvre, alors que cela aurait été souhaitable. Ensuite, cette mise en œuvre était inutilement complexe et difficile d'entretien.

Nous avons corrigé ce défaut avec Oto 2 en laissant la commande `rendre_tp` prendre

elle-même en charge ces validations. Les informations ne sont plus conservées dans un script, mais extraites du fichier d'informations utilisé également par la commande `decrire_boite`. Pour cette raison, il est maintenant possible de combiner ces options. Nous en avons également profité pour modifier la syntaxe des paramètres pour la rendre plus simple d'utilisation et plus efficace. L'exemple suivant montre comment créer une boîte à TP en spécifiant à la fois une date d'échéance et une liste exacte de travaux que les étudiants doivent déposer :

```
oto creer_boite --echeance="(23,59,2010,02,28)" --avec_liste_exacte='(fich1,fich2)'
```

4.5.3 Autres problèmes soulevés

L'intégration du DSL à Oto a été l'occasion pour nous de découvrir un certain nombre de problèmes, dont certains que nous avons abordés. Dans cette dernière section, nous traiterons d'autres problèmes soulevés lors de la mise en œuvre du DSL Oto et des solutions que nous avons apportées pour en venir à bout.

Dans les versions initiales d'Oto, une erreur dans le script (et non dans la correction effectuée par le script) survenant à la correction d'un groupe ne mettait pas fin à celle-ci ; l'erreur qui s'était produite était ajoutée au rapport et la correction reprenait au TP suivant. Une fois l'exécution de tous les travaux terminée, le rapport était ensuite affiché sur la sortie standard et c'était seulement à ce moment-là que le correcteur constatait la présence d'un problème. Une telle situation rendait particulièrement pénible le débogage de scripts Oto, car traiter tout le groupe pouvait être relativement long dans le cas de groupes de grande taille. Or, dans le cadre de nos modifications, nous avons modifié le comportement d'Oto pour qu'une assertion échouée mette immédiatement fin au script et affiche l'erreur appropriée sur le canal d'erreur.

Certains aspects des tests d'Oto ont été modifiés. L'un d'entre eux est l'utilisation d'expressions régulières pour chercher certains éléments présents dans les résultats remis par Oto. Or, de faux négatifs étaient provoqués par l'ajout du script utilisé pour le test correspondant dans le rapport complet de correction. Nous avons ajouté d'autres conditions pour éviter ces faux négatifs, notamment en recherchant des informations sur le TP avant la réponse attendue. Oto utilisant désormais un plus grand nombre de solutions de métaprogrammation dans son fonctionnement, il nous a été nécessaire de trouver une manière de tester ces nouveautés. Par exemple, les scripts Oto étant du code Ruby valide, il nous a été possible d'effectuer les tests de l'environnement d'exécution en exécutant, à l'intérieur même de cet environnement, des

assertions sur les résultats d'appels de modules et de commandes. Ces assertions étaient offertes par un objet d'une méthode de test, qui était visible depuis du code exécuté avec la méthode *instance_ eval* dans le même objet. Nous pouvions également encapsuler l'appel à cette méthode dans une assertion pour nous assurer quelles exceptions seraient soulevées par le bloc.

CHAPITRE V

EXEMPLES DES NOUVELLES CAPACITÉS DE CORRECTION

Comment pouvons-nous utiliser les nouvelles capacités de correction d'Oto avec de véritables cas? À tâche de correction comparable, à quel point une tâche de correction Oto 2 est-elle plus simple que son équivalent sous Oto 1+? Est-il possible d'intégrer en un seul fichier tout le nécessaire à une tâche de correxion complexe? Quels sont les inconvénients des nouveaux scripts Oto?

Dans ce chapitre, nous chercherons à comprendre à quel point les changements que nous avons apportées à Oto ont permis d'améliorer la flexibilité de la définition des tâches de correction. Pour parvenir à cette fin, nous examinerons plusieurs exemples d'utilisation d'Oto pour accomplir certaines tâches qui, sous Oto 1+, auraient exigé plusieurs niveaux de mise en œuvre et ou auraient même été virtuellement impossibles à réaliser. Nous commenterons chacun des cas pour mettre en lumière les avantages et, s'il y a lieu, les inconvénients de la nouvelle version.

5.1 Exemple 1 : intégration des commandes `bash` dans une vérification de remise

5.1.1 Présentation et description

Dans ce premier exemple, présenté à la figure 5.1, nous examinerons certaines des possibilités offertes par la manipulation des commandes dans les scripts Oto. Il s'agira d'un scénario de correction hypothétique où l'étudiant rend son TP sous la forme d'un fichier compressé.[1] Le

[1] Certains pourront remarquer, avec raison, qu'une telle situation serait peu probable dans les premiers cours de programmation, du moins à l'UQAM où l'utilisation de systèmes de type UNIX ne vient que plus tard

fichier sera décompressé, la présence de fichiers obligatoires sera vérifiée, notamment un fichier définissant une interface Java qui sera inspecté pour assurer que l'étudiant ne l'a pas modifié, afin de lui fournir un message d'erreur le cas échéant.[2]

Cet exemple tient compte de deux réalités propres aux vérifications exécutées lors de remises. D'abord, il ne sera pas possible de passer de paramètres aux scripts lors de la remise, car cela rendrait vraisemblablement trop complexe l'utilisation d'Oto par le ligne de commande pour les étudiants visés.[3] Ensuite, contrairement aux scripts de correction de groupe qui ne doivent pas échouer (devant traiter le groupe en entier, et non pas confirmer ou non la remise d'un seul travail), un script de remise doit soulever une exception si la remise ne doit pas être faite. Cette exception indiquera à la commande `rendre_tp` de rejeter la remise et d'afficher un message d'erreur dont nous lui en fournirons les détails.

5.1.2 Analyse de l'exemple

En utilisant plusieurs des capacités d'Oto et de Ruby pour effectuer une vérification avant remise complexe, l'exemple de la figure 5.1 montre à quel point les scripts Oto offrent une flexibilité accrue au niveau de l'appel de commandes `bash` et du traitement de leurs résultats. Bien qu'en théorie il aurait été possible d'avoir recours aux trois commandes dans les scripts d'Oto 1+, les appels nécessaires auraient pris plusieurs lignes chacun, le traitement des résultats n'aurait pas permis de manipuler des objets, comme la liste obtenue par la séparation des éléments de la sortie standard du `ls`. De plus, il aurait été fort difficile de trouver les chemins absolus et relatifs à utiliser pour l'accès aux fichiers à tester, Oto 1+ ne donnant pas facilement accès à cette information, contrairement au DSL Oto qui ajuste le répertoire courant au répertoire temporaire du TP en cours de correction.

---

au cours de leur cheminement.

[2]Toutes ces validations, et particulièrement celle de l'intégrité de l'interface, sont faites dans l'intérêt des étudiants. Après tout, les étudiants resteront toujours des étudiants... Pour une personne ne maîtrisant que très partiellement un langage de programmation et chez qui le rôle d'une interface n'a pas encore été totalement assimilé, il peut être tentant de modifier celle-ci pour qu'elle corresponde à l'idée que l'étudiant s'en fait, ce qui n'est évidemment pas le but de l'exercice.

[3]Nous avons eu l'occasion de constater que certains étudiants éprouvaient des difficultés à utiliser Oto, malgré des explications allant jusqu'à l'affichage de la commande Oto entière à saisir à la ligne de commande dans certains travaux du professeur Tremblay.

Un inconvénient de la syntaxe utilisée demeure que les assertions (supportées ici par le module `Pass`, facilitant l'utilisation des assertions, dont l'utilisation est présentée à l'appendice E) sont un peu lourdes d'utilisation et demandent un apprentissage des mécanismes de Ruby. Toutefois, nous estimons que cet apprentissage ne représente pas de difficultés particulières pour un programmeur accompli, notamment s'il connaît déjà Ruby.[4]

5.2 Exemple 2 : correction complexe avec paramètres et fichiers

5.2.1 Présentation et description

Nous présontons à la figure 5.2 un exemple de script Oto complexe recevant un paramètre depuis la ligne de commande, écrivant un fichier texte contenant le résultat du rapport de correction, mais également définissant une véritable méthode Ruby pour mieux structurer une partie du traitement qu'il effectue. Ce script est inspiré d'un véritable cas de correction qui nous a été fourni par le professeur Tremblay et dont nous avions déjà parlé à la section 4.2.6.3. Il sert à corriger des programmes parallèles en langage MPD. Dans cet exemple, la construction et l'appel des exécutables de tests est définie dans un `Makefile`. La tâche du script se limite à faire appel à la bonne version des cibles de l'outil `Make` telles que définies dans le fichier et à en colliger les résultats dans un fichier. Pour rendre les choses plus intéressantes, le professeur Tremblay ayant prévu deux séries de tests, une série publique et une privée, nous utiliserons un paramètre passé au DSL pour que la bonne sorte de tests soit choisie à l'exécution et le script adapté en conséquence. Une fois que la sorte de tests à exécuter est choisie, pour chacun des TP nous copions dans leur répertoire les fichiers fournis à l'activation de l'évaluation pour les tests[5], ceux-ci sont lancés et les résultats obtenus sont conservés sous des noms significatifs. Étant donné qu'il s'agit d'un langage parallèle, nous testons l'exécution avec divers nombres de processeurs disponibles ; ceux-ci sont définis dans une variable d'environnement. À la fin de la correction avec un nombre donné de processeurs, le rapport est produit. Aucun résultat n'est affiché à l'écran ; tout est écrit directement dans un fichier. Le script boucle sur le nombre de

---

[4]Il ne nous parait pas risqué d'affirmer que les développeurs de scripts Oto seront des programmeurs confirmés, puisqu'ils seront essentiellement professeurs ou chargés de cours, du fait vraisemblablement titulaires de maîtrises ou de doctorats en informatique. De plus, nous croyons que cet apprentissage serait également à la portée d'un auxilaire d'enseignement doué.

[5]Ou sur la ligne de commande en cas d'utilisation d'un fichier *.oto* employé seul.

```
nom_fichier_remise = 'tp2.tar.gz'
fichiers_attendus = [ 'tp2.java', 'Liste.java', 'maliste.java' ]

groupe.each { |tp|

  decompression = `gzip -dc #{nom_fichier_remise} | tar xf -`
  Pass.unless( decompression[:STDERR], :FICHIER_COMPRESSE_INCORRECT,
    :erreur => decompression[:STDERR],
    :cause_probable => 'Le fichier est compresse incorrectement ou corrompu.' )

  res_ls = `ls tp2`
  fichiers_remis = res_ls.split(' ')

  fichiers_attendus.each do |nom_fichier|
    Pass.if( fichiers_remis.include?( nom_fichier ), :FICHIER_NON_TROUVE,
      :fichiers_trouves => fichiers_remis,
      :fichiers_attendus => fichiers_attendus )
  end

  res_diff = `diff Liste.java ~/cours/java/tp2/Liste.java`
  Pass.unless( res_diff[:STDOUT], :INTERFACE_A_ETE_MODIFIEE,
    :cause_probable => "L'interface Liste.java a ete modifiee, alors que cela
    n'est pas permis. Veuillez la remplacer par le fichier original et compiler
    a nouveau votre travail avant de le rendre." )

  # Si on se rend ici, tout s'est bien passé.
  puts 'Votre travail a passé avec succès le test de remise.'

}
```

Figure 5.1 Exemple de script de vérification de remise avec plusieurs appels de commandes *bash*.

processeurs.

5.2.2 Analyse de l'exemple

La mise en œuvre d'une telle solution s'avère considérablement plus facile sous Oto 2 que dans les versions précédentes. Hormis le `Makefile`, aucun autre fichier que le script n'est nécessaire. Contrairement à Oto 1+, il n'est plus nécessaire d'utiliser un script `bash` pour, autant de fois qu'il y a de nombre de processeurs à tester, mettre à jour la variable d'environnement, exécuter Oto sur le groupe et rediriger la sortie standard vers un fichier. Il n'est plus nécessaire, par ailleurs, de prévoir deux scripts `Oto` différents pour les tests publics et privés. De trois fichiers, la coordination de la correction passe à un seul, qui plus est plus court, plus léger et plus simple. Par rapport à l'ancienne, la mise en œuvre de cet exemple ne semble posséder que des avantages. D'ailleurs, elle s'exécutera assurément plus rapidement, car le groupe ne sera chargé qu'une seule fois et Oto, tout comme le script, ne seront exécutés qu'une fois également. Nous examinerons la question de la performance au chapitre 6.

5.3 Exemple 3 : modules individuels et modules collectifs

5.3.1 Présentation et description

Dans ce dernier exemple, présenté à la figure 5.3, nous considérerons un scénario de correction combinant des appels de modules individuels et de modules collectifs. Celui-ci, inspiré de l'exemple de script Oto que nous avions présenté à la section 4.1.1, nous permettra de montrer comment les différents types de modules sont utilisés ensemble, ainsi que de présenter une autre fonctionnalité des résultats individuels permettant d'en hausser la flexibilité. Il s'agit d'un cas de correction conventionnel, compilation suivie d'exécution de tests. Selon les résultats obtenus, un message approprié sera affiché dans le rapport. Une note sera également attribuée en fonction du résultat. Ces notes seront colligées par le module de production de statistiques dont le résultat sera fourni dans le rapport.

5.3.2 Données de tests

Contrairement aux exemples précédents, celui-ci ne sera pas présenté seul, mais accompagné de deux extraits du rapport de correction ayant été affichés sur la sortie standard. Notre

```
if parametres['sorte'].nil? then
  nbAssertionsAttendues = 4791
  nom_sorte = 'publics'
else
  nbAssertionsAttendues = 4008
  nom_sorte = 'prives'
end
[1, 5].each do |nb_threads|
  if no_threads == 1 then
    ENV.delete('MPD_PARALLEL') if ENV.keys.include?('MPD_PARALLEL')
  else
    ENV['MPD_PARALLEL'] = nb_threads.to_s
  end
  groupe.each do |tp|
    rep_script.exporterVers tp.nom_repertoire
    `cp -f Tester-#{nom_sorte}.mpd Tester.mpd`
    ast = 0
    ast_inc = 0
    [0, 1, 2, 3].each do |nb_make|
      tp["Execution ##{nb_make} et #{nb_threads} thread(s)"] = make \
        "t#{nb_make} NBTHREADS=#{nb_threads}"
      str_stdout = tp["Execution ##{nb_make} et #{nb_threads} thread(s)"][:STDOUT]
      str_ast = str_stdout[/#{"tests,"}.*#{"assert. eval."}/]
      ast += str_ast.split("tests,")[1].split("assert. eval.")[0].strip.to_i \
        unless str_ast.nil?
      str_inc = str_stdout[/#{"echoues,"}.*#{"assert. inc."}/]
      ast_inc += str_inc.split("echoues,")[1].split("assert. inc.")[0].strip.to_i \
        unless str_ast.nil?
    end
    tp['Resultat sur 100'] = 100.0 * ( (ast - ast_inc) / nbAssertionsAttendues )
  end
  rapport = produire_rapport_complet( groupe )
  ecrire_fichier( "rapport-resultats-#{nom_sorte}-#{nb_threads}.txt", rapport )
  groupe.effacer_resultats    # Pour le prochain nombre de processeurs
end
```

Figure 5.2 Exemple de script de correction avec paramètre et écriture vers des fichiers.

correction a été faite sur quatre travaux, inspirés de l'exemple de l'appendice F du mémoire de maîtrise de Frédéric Guérin (17). En tout, six tests sont exécutés. Une note est attribuée en fonction du nombre de tests réussis, un travail ne compilant pas obtenant la note 0. Pour tester les différents cas possibles, un travail ne compile pas (obtenant 0 %), un autre échoue deux tests introspectifs (obtenant une note de quatre sur six, soit environ 66.667 %) et deux travaux réussissent tous les tests (et obtiennent 100 %).

5.3.3 Analyse de l'exemple

Les scripts d'Oto 2, comme ceux des versions initiales, permettent d'exécuter des modules et de choisir si leur contenu sera affiché ou non dans le script. Toutefois, Oto 2 va encore plus loin, car il permet de choisir, pour un résultat de module individuel, quelles seront les informations qui seront incluses. Par défaut, tout sera inclus, mais exclure certains identificateurs peut être utile lorsqu'on ne souhaite afficher qu'une partie du résultat, par exemple les détails d'une erreur détectée par un test introspectif. Pour ce faire, la méthode `inclure_resultat` permet de spécifier un identificateur ou une liste d'identificateurs qui doivent être inclus.[6]

Les extraits de rapports que nous présentons aux figures 5.4 (un extrait présentant les statistiques sur les notes obtenues durant la correction) et 5.5 (le résultat individuel obtenu par l'étudiant dont le travail ne compilait pas) affichent bien les informations attendues. D'abord, le module de production de statistiques s'acquitte normalement de son travail. Il s'agit d'un cas où un texte seul sera utilisé, le résultat n'étant pas accompagné d'une liste d'étudiants ayant réussi ou échoué, et le détail de chacun des étudiants ne contient pas de trace de ce module. Bien qu'il eut été théoriquement possible d'arriver à un résultat similaire avec les versions initiales d'Oto, colliger ces valeurs aurait exigé une manipulation complexe du rapport au moyen d'au moins un script `bash`. Nous pouvons constater que le second, présentant l'extrait consacré à l'étudiant dont le travail a échoué certains tests, illustre bien la limitation de la liste des résultats retournés par le module `tester_junit`, qui ne comprend que les détails retournés par le cadre de test, alors qu'elle comprendrait normalement plusieurs identificateurs contenant des informations sur les tests exécutés : nombre d'assertions, nombre d'assertions attendues, nombre d'erreurs, etc.

---

[6]La mise en œuvre de cette fonctionnalité a provoqué chez nous une réflexion. Devions-nous spécifier une liste de contenu à inclure, ou une liste contenant les identificateurs à exclure du résultat? Nous nous sommes finalement arrêtés sur une liste d'inclusion, car celle-ci nous semblait mieux respecter l'esprit du principe de moindre surprise de Ruby et du DSL Oto.

```
fclient = 'Client.java'
ctest = 'ClientTest'
note_max = 6.0
groupe.each { |tp|
  rep_script.exporterVers tp.repertoire
  res_javac = compiler_javac( tp ) {
    :fichiers >> fclient
  }
  if res_javac.reussi? then
    res_test = tester_junit( tp ) {
      :classe >> ctest
    }
    note = ( res_test[:nbtests] - res_test[:nberreurs] ).to_f
    if res_test[:nberreurs] > 0 then
      message = 'Le travail compile, mais contient une ou plusieurs erreurs.'
      res_test.inclure_resultat( :detail )
      tp['Resultats JUnit'] = res_test
    else
      message = 'Le travail ne contient aucune erreur. Felicitations!'
    end
  else
    tp[:Details] = res_javac.message_erreur_echoue
    message = 'Le travail ne compile pas.'
    note = 0
  end
  tp[:Message] = message
  # Note en pourcentage du nombre de tests réussis (affichée dans le rapport)
  tp['Resultat (%)'] = ( note * 100 ) / note_max
}

res_stats = produire_statistiques( groupe ) {
  :nom_variable >> 'Resultat (%)'
}
rapport = produire_rapport_complet( groupe, res_stats )
puts rapport
```

Figure 5.3 Exemple de script avec divers modules et inclusion partielle des résultats.

```
*********************************************************************

RESULTAT COLLECTIF: Statistiques sur les travaux

    Nombre de travaux consideres: 4
    Nombre de travaux dont le resultat 'Resultat (%)' n'etait pas defini: 0

    Pour le resultat 'Resultat (%)':

    Moyenne: 66.667
    Mediane: 83.334
    Ecart type: 47.14
    Resultat minimum: 0.0
    Resultat maximum: 100.0

    Distrubution des resultats:

                         100.0 => 2
                        66.667 => 1
                           0.0 => 1


*********************************************************************
```

Figure 5.4 Extrait de résultat du rapport complet de correction : statistiques.

```
*********************************************************************

TRAVAIL:   ad010101+2009.01.01.01.01.01.898567+abcd01010101.tp_oto

Equipe:     abcd01010101
Depot:      2009-01-01 a 01:01
Deposeur:   ad010101
Nom:        Etudiant Untel
Courriel:   untel.etudiant@courrier.uqam.ca


RESULTATS:

  Resultats JUnit: echoue
    detail:
      .F..F...
      Time: 0,003
      There were 2 failures:
      1) testConstructeur(ClientTest)junit.framework.AssertionFailedError: ...

      FAILURES!!!
      Tests run: 6,  Failures: 2,  Errors: 0

  Message:
      Le travail compile, mais contient une ou plusieurs erreurs.

  Resultat (%):
      66.6666666666667
```

Figure 5.5 Extrait de résultat du rapport complet de correction : résultat d'un étudiant.

CHAPITRE VI

ANALYSE DES PERFORMANCES

Quels gains de performance les modifications que nous avons apportées à Oto ont-elles entraînés, sur le même équipement? Les gains sont-ils proportionnels à la lenteur de la machine utilisée? Comment pouvons-nous expliquer ces gains? Le comportement des scripts et des modules a-t-il été amélioré? Existe-t-il des situations où la version 2 d'Oto sera plus lente que les versions initiales, et si oui, pourquoi?

Dans ce chapitre, nous traiterons d'abord de l'amélioration des performances d'Oto que la mise en œuvre du DSL Oto a apportée en comparant les temps d'exécution de la version 1+ à ceux de la version 2. Cette comparaison se fera sur plusieurs aspects de l'outil, notamment au cours de corrections et à l'exécution de ses tests intégrés. Ensuite, nous examinerons les causes de ces gains. Le passage à un paradigme de correction « de groupe » peut-il, à lui seul, expliquer ces gains ou les causes sont-elles partiellement autres? Finalement, nous chercherons à savoir si certaines commandes ou scénarios de correction s'exécutent plus lentement sous Oto 2 que sous Oto 1+. Le cas échéant, nous discuterons des causes possibles et des moyens d'y remédier.

6.1 Mesure de la performance

6.1.1 Comparaison des performances des versions d'Oto sur la machine `rayon1`

En premier lieu, nous testerons les différentes versions d'Oto sur le serveur de l'UQAM qui sera vraisemblablement utilisé pour supporter l'outil.[1] Celui-ci, au moment d'écrire ces lignes, était un Sun Fire X4150 de Sun Microsystems muni de deux processeurs Intel Xeon à quatre coeurs et de 16 gigaoctets de mémoire vive.

Nous reprendrons au tableau 6.1 une partie des cas de tests qui avaient été utilisés au chapitre 2 (à la section 2.4.2) pour tester les différences de performances entre Oto 1+ sous deux versions distinctes de Ruby, mais en exécutant tous ces tests sous Ruby 1.8.6. Encore ici, le meilleur temps en secondes sera en caractères gras.[2]

À l'exception de certains tests intégrés à Oto[3], la version 2 s'est révélée la plus rapide avec une marge variant selon la nature du test exécuté. Le test ayant recours au module de détection du plagiat s'est exécuté avec une marge encore plus grande, démontrant les avantages du nouveau paradigme de correction au niveau de la correction de groupe. Le scénario de correction pour INF3140, ayant recours aux commandes `bash`, sera aussi accéléré en diminuant de deux à un seul les instances d'Oto nécessaires à la correction et en éliminant l'appel de modules pour l'exécution des commandes. Un gain d'une dizaines de secondes sur une machine

---

[1] Comme nous l'avons mentionné au chapitre 2, Oto avait été initialement conçu pour la machine Arabica de l'UQAM. Toutefois, les machines Rayon (rayon1 et rayon2) sont plus modernes et plus performantes. Nous verrons les différences de performances d'Oto entre ces machines plus loin dans ce chapitre.

[2] Le lecteur aura aussi remarqué que nous n'utilisons pas pour les fins de cette section la machine qui avait été utilisée à la section 2.4.2, car nous n'avions pas besoin ici d'une version de Ruby non supportée par l'UQAM et souhaitions que les temps obtenus se rapprochent autant que possible de l'expérience à laquelle un utilisateur d'Oto pourra s'attendre. Pour des raisons de différences technologiques, les temps présentés ici seront parfois moins rapides que ceux de la machine utilisée au chapitre 2. Nous croyons toutefois que la comparaison significative sera celle effectuée sur une même machine.

[3] Les tests en question ont été convertis pour faire appel à un démarrage complet d'une autre instance d'Oto, ce qui en fait un test plus près d'un scénario de correction réel. Comme plusieurs fichiers doivent être chargés et interprétés sur disque, auquel l'accès est coûteux, le test sera généralement plus lent. De plus, `tester_junit` comprend plus de tests que le module `junit` d'Oto 1+, car il intègre les fonctionnalités de JUnit 4, lesquelles étaient supportées par un module séparé dans Oto 1+.

Test	**Oto 1+**	**Oto 2**	**% de gain**
Démarrage d'Oto sans arguments	0,45	**0,19**	57 %
Test de la commande `rendre_tp`	**98,22**	101,36	-3 %
Test du module `compiler_javac`	29,61	**25,07**	15 %
Test du module `tester_junit`	**24,66**	33,28	-35 %
`verifier_tp`, `compiler_javac` et `tester_junit`	5,09	**2,63**	48 %
`corriger_groupe`, `compiler_javac` et `tester_junit`	9,23	**6,82**	26 %
`corriger_groupe` et `detecter_plagiat`	55,24	**28,24**	49 %
`corriger_groupe` complexe pour INF3140	103,39	**93,20**	9 %

Tableau 6.1 Temps d'exécution de tests selon la version d'Oto sur la machine `rayon1`.

performante contribue à démontrer l'efficacité de la nouvelle version.

6.1.2 Scénario de correction complet pour le cours INF5170

Pour vérifier les performances d'Oto dans le cas d'un scénario de correction complexe et d'un groupe de grande taille, nous avons eu recours à de véritables travaux remis au professeur Tremblay dans le cadre du cours INF5170 de l'UQAM à la session d'automne 2008. Ces travaux devaient subir deux corrections différentes. D'abord, nous devions assurer leur bon fonctionnement en comparant les résultats qu'ils retournaient aux résultats attendus. Ensuite, comme il s'agissait d'un travail de programmation parallèle, nous devions vérifier le temps nécessaire à l'exécution des différents cas pour une quantité variable de processeurs disponibles. Ce dernier test, appliqué au groupe entier, était d'autant plus lourd que certains travaux moins bien construits prenaient un temps considérable à s'exécuter. Les résultats de l'exécution de ces tests sur la machine Arabica[4] sont présentés au tableau 6.2.

Les deux tests donnent gain de cause à Oto 2 au niveau des temps d'exécution. Si les tests de vitesse s'exécutent plus rapidement d'environ 6 minutes sur Oto 2, ce qui représente un certain gain, son véritable avantage consiste à exécuter plus rapidement un nombre élevé de

[4]Au moment de rédiger ce mémoire, MPD, le langage utilisé dans ce travail pratique, ne pouvait exploiter adéquatement que le machine Arabica, disposant de six processeurs physiques. Les faibles performances de cette machine dûes à son âge vénérable expliquent en partie les résultats présentés ici.

Correction	Oto 1+	Oto 2
Résultats	922,46	**411,27**
Vitesse	7206,00	**6840,00**

Tableau 6.2 Temps de correction en secondes de travaux du cours INF5170 (Arabica).

tests de courte durée appelés au moyen du logiciel `Make`, où le temps de chargement du module `cmd_bash` handicape considérablement Oto 1+. Les temps du test de résultats, qui donnent un net avantage à Oto 2, témoignent du remplacement de ce module par un mécanisme plus simple.

6.1.3 Comparaison des versions d'Oto selon le serveur utilisé

Dans cette dernière comparaison, nous avons exécuté l'ensemble de la chaîne de tests d'Oto des différentes versions sur les serveurs supportés par l'outil. Les temps d'exécution sont rapportés au tableau 6.3. Si nous croyons que de vrais scénarios de correction sont plus représentatifs des gains de performance obtenus avec Oto 2 [5], nous avons néanmoins inclus cette comparaison pour confirmer que les gains ont été vérifiés sur toutes les machines, et ne seront pas causées par les caractéristiques particulières que pourraient posséder certaines de celles-ci.

Serveur	Oto 1+	Oto 2
Arabica	1638,82	**1521,90**
Rayon1	496,65	**383,98**
Zeta	1304,12	**1236,65**

Tableau 6.3 Temps d'exécution en secondes de la chaîne de tests selon le serveur utilisé.

Comme nous pouvions nous y attendre, les chiffres favorisent Oto 2 sur toutes les machines, avec un gain plus significatif pour Rayon1. Les causes de cet avantage iraient, à notre avis, de la plus grande quantité de mémoire vive disponible sur cette machine qui diminue le recours à

[5] Il est en effet difficile de comparer les tests des différentes versions d'Oto, les composants et la quantité de fichiers devant être testés n'étant pas les mêmes. De plus, l'utilisation directe de fichiers *.oto* dans les tests a tendance à désavantager Oto 2, car elle devra activer et désactiver une évaluation par test, alors qu'Oto 1+ pouvait réutiliser une évaluation pour plusieurs tests (dans ce cas, l'utilisation de fichiers *.oto* se justifie par la simplification des cas de tests).

la mémoire virtuelle.

6.2 Analyse des résultats

Comme nous l'avons constaté, à l'exception de certains cas de tests intégrés, où le nombre et la nature des tests ont été considérablement modifiés d'une version à l'autre, Oto 2 s'est révélé plus rapide que son prédécesseur dans toutes les situations d'utilisation. À ce niveau, l'écart obtenu dépendait de la nature de la tâche. Les résultats obtenus à la section 6.1.2 démontrent les différences de gains en fonction du surcoût associé à l'utilisation d'Oto par rapport à un appel manuel des tâches de correction.[6] Contrairement aux versions initiales, Oto 2 ne charge et n'exécute le script qu'une seule fois par instance, réduisant d'autant le surcoût associé à de courtes tâches de correction répétées, par exemple de courts appels de commandes. Par contre, l'outil lui-même ne peut réduire le temps nécessaire à l'exécution d'une tâche de correction qui serait intrinsèquement longue, comme dans l'exemple du test de vitesse, où la vaste majorité du temps était passé à exécuter les programmes des étudiants activés par une commande *bash*. Le passage à Oto 2 a toutefois permis, dans ce cas, d'accélérer légèrement la correction par une réduction du surcoût (au niveau de l'appel des commandes), mais ne peut l'accélérer de manière plus significative sans altérer la tâche elle-même.

Au niveau de la quantité de travail faite par l'application, certains aspects ont été améliorés par la nouvelle version. L'exécution des scripts, par exemple, ne se fait qu'une seule fois, au lieu d'autant de fois qu'il y a de TP à corriger. Bien que la quantité de travail à effectuer demeure la même au niveau des tâches de correction individuelles, l'utilisation de modules collectifs est nettement avantagée, réduisant à une seule fois le chargement du module, la validation des paramètres et le retour des résultats obtenus. Les modules collectifs étant désormais supportés par l'outil de manière native, celui-ci n'est plus ralenti par l'application du subterfuge que nous avions utilisé pour les faire fonctionner dans les versions initiales (voir la section 2.3.2.3). De plus, l'augmentation des capacités de correction, notamment en permettant au script d'écrire dans des fichiers, réduit d'autant le nombre de niveaux d'applications nécessaires à la mise en œuvre, ce qui contribue à hausser ses performances.

Mêmes certains aspects insoupçonnés de l'application peuvent bénéficier des modifications

[6] Tel que le chargement d'Oto en mémoire, le parcours de ses fichiers, l'appel de la commande à utiliser, le mécanisme d'exécution du script, etc.

que nous lui avons apportées en terme de performances, notamment les erreurs dans les scripts Oto. Là où les versions initiales exécutaient un script comprenant une erreur à l'exécution de ce script[7] une fois par TP, ignorant le fait qu'une erreur s'était produite, Oto 2 arrête immédiatement l'exécution de la correction. Cette capacité de propagation des erreurs facilite le débogage des scripts.

De manière générale, les temps d'exécution que nous avons obtenus nous sont apparus raisonnables et acceptables, davantage que ceux de ses prédécesseurs. À l'exception de la correction de vitesse, dont le temps dépend principalement des temps requis par l'exécution des programmes des étudiants, l'ensemble des résultats obtenus furent produits en au plus quelques minutes. Une vérification de TP individuelle se fait en une seconde ou deux, ce qui est un temps acceptable en laboratoire. Pour ce qui est des correcteurs, nous avons réduit tant le temps nécessaire à la correction au total que le délai avant la mise en évidence d'une erreur dans un script, ce qui représente un progrès intéressant.

6.3 Améliorations possibles

Un des points les plus importants que notre travail sur Oto a pu démontrer demeure que les performances d'un outil d'aide à la correction seront tributaires de la nature de la tâche à effectuer. Pour cette raison, des travaux futurs visant à améliorer les performances d'Oto pourraient reposer davantage sur le développement de modules plus efficaces et plus performants en prenant notamment avantage des nouvelles capacités de correction de l'outil et du paradigme de correction de groupe. Il serait possible d'imaginer, par exemple, un module de compilation Java collectif qui ne démarrerait la machine virtuelle qu'une seule fois pour traiter un groupe entier, notamment grâce à l'API de compilation de Java 6 et ultérieures. Nous pourrions également utiliser une telle approche pour un module JUnit collectif.

[7] Et non pas une erreur dans le programme étudiant exécuté par ce script.

CONCLUSION

Un outil d'aide à la correction des travaux de programmation peut représenter, pour le correcteur, un atout lui permettant d'effectuer un travail de meilleur qualité, et de le faire plus rapidement que s'il devait procéder à une correction entièrement manuelle. C'est pour cette raison que depuis plusieurs décennies, un certain nombre d'outils destinés à cette fin ont vu le jour. Malgré leur potentiel, ceux-ni n'ont pas toujours connu un grand succès, étant limités dans leurs capacités et souvent destinés à une seule tâche de correction en particulier. Le projet Oto visait à corriger ce défaut, se voulant dès le départ générique et extensible. Si le principe autour duquel Oto a été construit était bon, certains aspects de sa mise en œuvre nuisaient à son utilisation en situation réelle. C'est pour cette raison que le professeur Tremblay, notre directeur de recherche, a cherché à modifier Oto pour le rendre plus flexible, c'est à dire augmenter ses capacités de correction, et en hausser les performances, pour qu'il soit utilisable tant pour un correcteur ayant une tâche complexe à exécuter que par un étudiant ayant une correction simple à tester dans le cadre d'un laboratoire limité dans le temps.

Notre analyse de l'outil nous a permis de comprendre certains principes inhérents aux outils de correction. Le plus important dans le cas d'Oto est qu'un outil ne pourra s'exécuter plus rapidement que le temps nécessaire à l'exécution des tâches effectuées pour réaliser la correction des TP. Notre tâche devait donc se concentrer sur la diminution du surcoût entraîné par l'utilisation d'Oto pour manipuler les logiciels utilisés pour la correction : compilateurs, cadres de tests, etc. En même temps, bien que l'emploi de modules pour la mise en œuvre des capacités de correction lui permettait de demeurer générique, un module était souvent nécessaire là où une solution plus simple telle qu'une commande *bash* aurait suffit. De plus, la mise en œuvre d'un module, qui à l'origine devait être également à la portée de l'utilisateur d'Oto, s'est avérée plus compliquée et fastidieuse que prévu. Bien que le professeur Tremblay ait mis au point un module permettant de lancer une commande quelconque, celui-ci était lourd à utiliser. De plus, les scripts eux-même ne fournissaient pas à l'utilisateur un degré de contrôle très grand sur leur exécution et étaient limités dans leurs capacités, ne disposant que d'un ensemble restreint de structures. Toutefois, le principal problème était qu'Oto ne possédait pas une vision de groupe : tous les travaux étaient considérés de manière indépendante. Un tel paradigme handicapait à

la fois la flexibilité de l'outil, limitant la correction intra-groupe, et diminuait les performances, car le même script était chargé et interprété autant de fois qu'il y avait de travaux à corriger dans un groupe.

La solution que nous avons proposée est venue combler une majeure partie des exigences de flexibilité et de performances d'Oto, haussant considérablement ses capacités en terme de correction, simplifiant le développement de scripts Oto, le tout en s'exécutant plus rapidement que les versions initiales. Un DSL interne venant s'intégrer au langage Ruby dans lequel Oto était développé diminuait la complexité de la mise en œuvre, car il nous offrait ses structures et ses capacités sans que nous ayons à déployer d'efforts particuliers. Cette gratuité avait toutefois un prix : la syntaxe du langage que nous allions mettre en œuvre devait être du code Ruby valide. Cette obligation pouvait cependant être vue comme un avantage, car il nous paraissait plus facile de convaincre un utilisateur potentiel d'Oto d'imposer un apprentissage préalable s'il pouvait réutiliser ses connaissances avec des programmes Ruby en général que si celles-ci étaient limitées au seul outil (comme OtoScript). En bâtissant Oto 2 autour du DSL Oto, nous avons pu hausser considérablement les capacités des scripts.

Nous avons aussi augmenté la flexibilité des scripts en découplant la production de rapports de correction, permettant de décrire les résultats obtenus à l'exécution des commandes utilisées pour lancer la correction et en offrant la capacité, entre autres, de les sauvegarder sur disque au lieu de les afficher sur la sortie standard de manière systématique. Nous avons également ajouté un support amélioré pour les commandes, simplifiant considérablement leur utilisation ainsi que la récupération de leurs résultats. Nous avons également diminué le surcoût associé à leur utilisation en ne les associant plus à des appels de modules. Cette modification a également été pour nous l'occasion de permettre à Oto de supporter les corrections intra-groupe. Nous avons démontré ces dernières en mettant au point deux modules collectifs de correction qu'il n'aurait pas été possible de mettre en œuvre proprement dans les versions initiales d'Oto.

Les résultats que nous avons obtenus nous permettent de conclure, de manière générale, à une réussite. Les scripts Oto sont plus souples, plus puissants et plus simples à utiliser pour quiconque connaît le langage Ruby. Ils permettent d'effectuer de manière naturelle plusieurs tâches qu'il était possible d'effectuer avec Oto 1+, mais qui exigeaient une manipulation complexe des commandes de l'outil et de scripts *C-Shell*. Oto 2 est également plus rapide que son prédécesseur, particulièrement lorsqu'un script contient un grand nombre de courtes tâches. Il permet également de mettre en lumière une erreur éventuelle dans un script plus rapidement que

les versions initiales, car il cessera immédiatement de s'exécuter au lieu de lancer potentiellement en vain la correction du reste du groupe. Bien qu'une expérimentation en laboratoire réel reste à faire, nous croyons que les modifications que nous avons apportées à l'outil vont permettre son utilisation dans les situations pour lesquelles l'outil avait été initialement conçu.

Les langages spécifiques au domaine, déjà très employés dans l'industrie, semblent offrir une avenue de recherche qui ira encore davantage de l'avant dans les années à venir. Les DSL en général permettent de mieux cerner les besoins d'un domaine particulier. L'idée d'un DSL est assimilable à la construction préalable d'outils spécialisés pour réaliser une œuvre au lieu de se lancer directement dans sa réalisation avec des outils génériques moins adaptés. Faite correctement, elle ouvre la porte à la réalisation de la tâche par les spécialistes du domaine au lieu des seuls professionnels de l'informatique, ou à une hausse de la productivité de ces derniers. Toutefois, le coût de mise en œuvre d'un DSL externe tel qu'OtoScript, bien que permettant habituellement de mieux cerner la syntaxe spécifique de son domaine, peut être désavantageux, d'où l'intérêt d'un DSL interne où le développeur n'aura pas à mettre en place un compilateur et/ou un interprète. En développant un DSL interne destiné à un outil de correction des travaux de programmation, ce qui, au meilleur de nos connaissances n'avait jamais été fait auparavant, nous avons été à même de comprendre pourquoi ce sujet fait couler tant d'encre depuis quelques années.

Notre solution n'est toutefois pas sans défauts. Les DSL internes sont particulièrement sensibles aux erreurs syntaxiques, lexicales et aux problèmes survenant à l'exécution, le DSL Oto ne faisant pas exception. Une telle sensibilité demande de l'utilisateur une connaissance préalable de la syntaxe et des capacités du langage pour être en mesure d'identifier ses erreurs, ce qui augmente quelque peu la courbe d'apprentissage nécessaire à l'utilisation des scripts Oto. À ce niveau, la prise en mains d'Oto 2 demandera sans doute autant d'efforts que celle qui était nécessaire pour les versions initiales, mais la possibilité d'utiliser les connaissances ainsi acquises en dehors du contexte d'Oto jouent en sa faveur.

Il va de soi que plusieurs autres aspects d'Oto pourraient bénéficier d'améliorations. La question des messages d'erreur peu significatifs pourrait être abordée. Au niveau des autres fonctionnalités de l'outil, bien que nous en ayons amélioré les capacités de correction, Oto exécute toujours celles-ci dans le compte et avec les privilèges de son utilisateur, ce qui l'expose à tout

ce qui pourrait se trouver dans les TP.[8] Idéalement, la correction devrait être effectuée dans un espace isolé et protégé, comme dans une machine virtuelle. Parions que cela représenterait un défi intéressant. Il serait aussi utile à Oto de disposer de nouveaux modules et rapports pour étendre ses capacités. Pensons à un rapport qui enverrait ses résultats directement aux étudiants par courrier électronique, ou encore qui irait les déposer dans une base de données, ou dans les applications `Résultats` ou `Moodle`. Un module pourrait permettre d'attribuer des notes littérales en fonction des résultats et de barèmes établis, ou encore analyser la qualité du code des étudiants. À ce niveau, les seules limites semblent celles de notre imagination.

Pour conclure ce mémoire, nous traiterons brièvement de l'influence qu'a eu ce projet sur nos compétences académiques et professionnelles. Nos études de maîtrise en informatique furent pour nous l'occasion de découvrir le milieu de la recherche. Nous avons été amenés à consulter la littérature scientifique en lisant plusieurs articles de recherche, tant dans le cadre des cours que nous avons suivis que de ce projet. L'analyse d'Oto a été l'occasion pour nous de comprendre le fonctionnement d'une application qui n'était pas triviale, développée dans un langage que nous ne connaissions pas. Nous nous sommes habitués, au contact de notre directeur de recherche, à justifier nos prises de position de manière précise et détaillée ainsi qu'à accepter ses conseils et ses critiques constructives. Cette recherche fut pour nous l'occasion d'enrichir considérablement nos connaissances en informatique et en génie logiciel dans le cadre d'un projet de plus longue haleine que ceux auxquels nous avions été habitués. Par dessus tout, ces études de maîtrise nous ont permis d'acquérir une plus grande confiance en nous-mêmes et en nos capacités professionnelles.

Finalement, notons qu'en ce qui nous concerne, l'aventure « Oto » n'a pas débuté en entreprenant nos études de maîtrise, mais dès l'hiver 2006, lorsque nous avons assisté à une présentation du projet donnée par le professeur Tremblay. À ce moment, se doutait-il que dans la salle se trouvait un étudiant de première année du baccalauréat en informatique et génie logiciel, fasciné (à notre grande surprise) par le sujet abordé, qui saisirait plus tard l'occasion de poursuivre ce projet? Pour nous, revenir sur cela en écrivant les derniers mots de ce mémoire après avoir apporté notre contribution personnelle à Oto se fait avec une certaine nostalgie.

---

[8] Imaginons qu'un étudiant un peu espiègle décide de glisser un juteux « `rm -rf ~` » dans son TP...

APPENDICE A

HISTORIQUE DES VERSIONS D'OTO

Dans cet appendice, nous listerons les commandes et les modules qui furent ajoutés à chaque version d'Oto. Nous détaillerons également les particularités de chaque version.

A.1 Oto 1

Oto 1 était la toute première version d'Oto. Elle a surtout permis de mettre en œuvre l'architecture de l'outil et de prouver son concept. Pour cette raison, elle ne contenait qu'un minimum de commandes et deux modules. Dans cette première version, les scripts étaient développés dans le langage OtoScript, traduits par le traducteur Oto vers des fichiers images, puis interprétés par le moteur et le contexte d'exécution d'Oto. La correction se faisait de manière isolée, un seul TP à la fois.

Liste des commandes :

- `corriger_groupe` : Corrige un groupe de travaux par rapport à une évaluation.
- `creer_boite` : Crée une boîte à TP.
- `detruire_boite` : Détruit une boîte à TP.
- `detruire_espace` : Détruit tout l'espace Oto de l'utilisateur : boîtes et évaluations.
- `prendre_tp` : Permet au correcteur de prendre les travaux remis dans une boîte en vue de la correction.
- `publier_eval` : Permet d'activer une évaluation.
- `rendre_tp` : Permet à l'étudiant de rendre un TP.

- retirer_eval : Désactive une évaluation existante.
- tester_module : Lance des tests internes à Oto.
- verifier_tp : Corrige un seul TP par rapport à une évaluation.

Liste des modules :

- javac : Compilation de fichiers Java.
- junit : Exécution de tests introspectifs pour des classes Java.

A.2 Oto 1+

Oto 1+ est une version augmentée d'Oto 1. Cette augmentation signifie qu'elle dispose de davantage de fonctionnalités, mais que celles-ci ne viennent pas affecter les principes et l'architecture du logiciel. La plupart de ces améliorations ont été développées au fil du temps par le professeur Tremblay, mais les commandes afficher_evaluation, decrire_boite ainsi que le module plagiat ont été réalisés par l'auteur de ces lignes. Les fonctionnalités additionnelles se sont manifestées essentiellement par de nouvelles commandes ainsi que par le développement de modules d'extension supplémentaires.[1] Certaines commandes ont également été modifées, notamment creer_boite qui peut spécifier une date limite de remise et une liste de fichiers devant être remis au lieu d'une évaluation de remise. Elle permet également d'exécuter un script à la remise, dont l'échec entraîne l'annulation de la remise.

Ajout des commandes :

- afficher_evaluation : Affiche le contenu d'une évaluation : script et fichiers.
- confirmer_remise : Confirme à un étudiant la remise de son TP dans une boîte.
- creer_et_activer_eval_classe : Crée des tests introspectifs sur des classes à partir d'un fichier modèle en une seule commande.

[1] Notons que certaines commandes, telles que decrire_boite, ont été ajoutées au cours du développement d'Oto 2. Alors que nous aurions pu les ajouter à la seule nouvelle version, nous avons préféré les mettre immédiatement en production avec Oto 1+, car leurs ajouts étaient immédiatement utiles. C'est pour cette raison qu'Oto 2 n'introduit pas de nouvelle commandes.

- `creer_et_activer_eval_filtre` : Crée des tests de type entrées/sorties à partir d'un fichier modèle en une seule commande.

- `creer_et_activer_eval_methodes` : Crée des tests introspectifs sur des méthodes à partir d'un fichier modèle en une seule commande.

- `decrire_boite` : Permet de décrire les caractéristiques d'une boîte et de lister les TP qui y ont été remis.

- `decrire_module` : Affiche la description d'un module.

- `lister_boites` : Liste les boîtes d'un utilisateur.

- `lister_commandes` : Affiche la liste de commandes Oto disponibles. Similaire à lancer Oto sans commande ni script Oto.

- `lister_evaluations` : Liste les évaluations d'un utilisateur.

- `supprimer_doublons` : Permet de supprimer les doublons parmi les TP remis. Les doublons sont les TP multiples remis par une même équipe.

- `tester_classe` : Comme `creer_et_activer_eval_classe`, mais exécute directement l'évaluation sur un ou plusieurs TP.

- `tester_filtre` : Comme `creer_et_activer_eval_filtre`, mais exécute directement l'évaluation sur un ou plusieurs TP.

- `tester_methodes` : Comme `creer_et_activer_eval_methodes`, mais exécute directement l'évaluation sur un ou plusieurs TP.

Ajout des modules :

- `cmd_bash` : Lancement d'une commande `bash` quelconque et récupération des résultats obtenus.

- `gcc` : Compilation de programmes C et C++ par le compilateur C GNU.

- `junit4` : Exécution de tests introspectifs pour des classes Java avec JUnit 4, les différences entre les versions étant significatives, particulièrement au niveau de la syntaxe d'appel de tests.

- `plagiat` : Détection du plagiat intra-groupe.
- `tester_filtre` : Exécution de tests basés sur le filtrage des entrées/sorties des programmes cibles.

A.3 Oto 2

Oto 2 est la version d'Oto à partir de laquelle les scripts Oto ne sont plus définis en OtoScript, mais en Ruby valide interprété dans le contexte du DSL Oto. Si les commandes n'ont pas été fondamentalement affectées par ces modifications majeures, à l'exception, majoritairement, des commandes exécutant des évaluations, nous avons profité de l'occasion pour corriger certaines limitations, notamment celles de `creer_boite` en lui permettant de combiner l'utilisation d'évaluations de remise, la présence d'une date limite de remise ainsi qu'une liste de fichiers à remettre. Les modules existants ont été adaptés aux changements de la nouvelle version (désormais divisés entre modules individuels et modules collectifs) et renommés pour mettre davantage leur rôle en évidence. Certains modules furent supprimés, leurs fonctionnalités étant intégrés à d'autres ou carrément rendues caduques. Les noms des modules selon les versions sont détaillés au tableau A.1. Oto 2 introduit également le concept de rapports, qui viennent augmenter la modularité du résultat pouvant être affiché à l'exécution du script.

Ajout des modules :

- `produire_statistiques` : Module collectif permettant de colliger des statistiques sur une note définie dans un groupe. Celle-ci doit être une valeur numérique.

Ajout des rapports :

- `produire_rapport_complet` : Affichage de toutes les informations disponibles dans les résultats fournis au rapport, en plus du nom de l'évaluation, de la date de l'exécution et du script Oto utilisé pour définir l'évaluation.

Nom sous Oto 1+	**Nom sous Oto 2**
cmd_bash	*Supprimé* (appel de commandes intégré au langage)
gcc	compiler_gcc
javac	compiler_javac
junit	tester_junit
junit4	*Supprimé* (fusionné avec junit)
plagiat	detecter_plagiat
	produire_statistiques
tester_filtre	tester_filtre

Tableau A.1 Noms des modules selon la version d'Oto.

APPENDICE B

RAPPORT INTERNE : VERS LE DÉVELOPPEMENT D'UN MODULE DE DÉTECTION DU PLAGIAT POUR OTO

Ce rapport a été rédigé en juin 2008, avant la mise au point d'un module de détection du plagiat intra-groupe. L'idée était de comprendre la problématique du plagiat en milieu universitaire, de considérer les diverses mises en œuvre possibles d'un tel module et les logiciels externes que nous pourrions utiliser à cette fin. Ce sont sur les bases de celui-ci que nous avons développé le module `detecter_plagiat` autour du logiciel SIM.

B.1 Introduction

Le problème de la malhonnêteté intellectuelle, s'il n'est pas propre aux institutions d'enseignement, demande de leur part une attention particulière, étant chargées de discerner les étudiants maîtrisant la matière enseignée de ceux qui ne la maîtrisent pas. Dans le cas d'un travail de programmation, les possibilités de plagier sont d'autant plus importantes qu'il semble facile de faire une copie du travail d'autrui et de tenter de camoufler les ressemblances avec celui-ci tout en conservant ses fonctionnalités et résultats à l'exécution. La taille des groupes étant souvent élevée, particulièremement dans les cours d'introduction, le temps et l'énergie nécessaires à une correction manuelle de travaux se ressemblant tous rendent difficile la détection des cas de plagiat. Sachant que la tentation de plagier peut être d'autant plus grande que le risque que ce plagiat soit détecté et réprimandé est perçu comme étant faible,[1] le problème de la malhonnêteté prendrait, selon certains chercheurs, des proportions endémiques. Par exemple,

[1]Peut-être avec raison, dans les circonstances actuelles... Sans prétendre que la supposition d'une impunité probable face au plagiat ferait partie de l'*habitus* de l'étudiant universitaire (pour citer Bourdieu), supposons que la tentation sera forte en l'absence d'une détection rigoureuse, systématique et médiatisée du plagiat.

l'université nationale de Singapour a détecté en l'an 2000 dans le cadre d'un premier travail pratique de programmation qu'un nombre aussi important que 98 travaux sur 712 étaient plagiés, soit prés de 15 % (7).

Face à cette possibilité, l'intérêt d'un outil permettant de détecter automatiquement les cas de plagiat apparaît clairement. Cette idée n'est certes pas nouvelle, la littérature nous permettant d'identifier plusieurs outils voués à cette fin: YAP (38), Moss (27), SIM (15), JPlag (26) et autres Roboprof (8) parmi plusieurs, dont les algorithmes ont chacun des forces et des faiblesses. Plus près de nous, faisons mention d'un rapport informel rédigé par notre directeur de recherche il y a déjà presque 10 ans (36), où il exposait les caractéristiques de plusieurs outils qui existaient à ce moment, et exprimait le souhait que l'UQAM dispose de logiciels destinés à l'automatisation de la correction des travaux de programmation et de la détection du plagiat. L'outil Oto (17; 33) étant déjà venu répondre en partie à l'automatisation de la correction, il serait souhaitable d'y ajouter la capacité de déceler partiellement les cas de tricherie. Étant conscients de l'existence de divers outils, nous ne souhaitons pas tenter de réinventer la roue, mais trouver une manière de les exploiter de façon optimale et la plus simple possible en ajoutant à Oto un module de détection du plagiat (ce qui était même suggéré par son auteur principal dans son mémoire de maîtrise). L'objectif principal de cette fonctionnalité serait d'appliquer un certain nombre d'heuristiques aux travaux remis par les étudiants pour mettre en lumière des cas jugés suspects, qui seraient ensuite vérifiés manuellement par le correcteur. Cet objectif devra être atteint en réduisant le plus possible le nombre de faux positifs, mais en demeurant suffisamment sensible pour s'avérer fonctionnel en pratique.

B.2 Comment les étudiants peuvent-ils tricher?

Mike Joy, dans un article paru en 1999 (21), range les modifications apportées par les étudiants à un code source plagié (dans le but de masquer ce plagiat) en deux catégories. Il va de soi que nous souhaiterions que toutes deux soient détectables par notre module. Au niveau du premier type, les changements lexicaux, l'auteur explique que des commentaires peuvent être modifiés, ajouté ou supprimés, l'indentation et le nom des divers identifiants peuvent être changés, de même que les numéros des lignes, lorsque le langage le supporte (e.g. Fortran). Joy poursuit en décrivant un second type, les changements structurels possibles: les types de boucles peuvent être changés (de `while` à `for`, par exemple), une série de `if` imbriqués peut être remplacée par des `cases`, l'ordre des instructions et des déclarations peut être changé, si ce

changement n'affecte pas le fonctionnement du programme, les appels de procédure peuvent être remplacés par des appels de fonctions et vice-versa, l'appel d'une procédure peut être remplacé par le corps de la procédure et l'ordre des opérandes d'une expression peut être remplacé, en autant que la résultante soit algébriquement équivalente à l'originale.

Nous croyons (craignons ?) que la plupart des tactiques utilisées par les plagiaires, tactiques que nous avons citées plus haut, puissent être appliquées avec une aisance relative, au moyen d'un simple éditeur de texte et soient à la portée d'étudiants débutant en programmation, possiblement à partir de la copie d'un voisin de clavier plus avancé.[2] Il serait aussi possible d'imaginer le cas d'un étudiant qui découvrirait la copie d'un collègue sur un poste de travail commun et, peu scrupuleux, déciderait de la reprendre comme base pour son propre travail.[3] D'ailleurs, le lecteur intéressé aux tactiques de plagiat pourra se référer à l'article *Patterns of plagiarism* (8), qui présente plusieurs autres cas intéressants, que nous ne détaillerons pas ici pour ne pas alourdir ce texte.

B.3 Intégrer la détection du plagiat à Oto

Oto, qui est un outil générique et extensible pour corriger les travaux de programmation, pourrait gagner à disposer d'un module de détection du plagiat, que le correcteur pourrait intégrer à ses vérifications privées. Pour ce faire, il lui suffirait d'ajouter l'appel au module dans le fichier OtoScript correspondant à son correcteur privé et de lui fournir les paramètres appropriés.[4] À l'instar de son utilisation, le fonctionnement du module de détection du plagiat serait relativement simple.[5] Celui-ci n'implémenterait pas lui-même la logique nécessaire à la détection du plagiat, mais procéderait à toutes les tâches nécessaires à l'appel d'un ou de plusieurs

---

[2] Ou même que la copie soit modifiée par l'auteur du programme original...

[3] Nous devons cependant noter ici qu'il ne s'agit pas forcément d'une maladresse de la part de celui ayant fait l'oubli. L'un des outils employés dans le cadre des premiers cours de programmation, BlueJ, rend particulièrement difficile l'assurance que les fichiers d'un projet soient bel et bien effacés. De plus, il ne serait pas inimaginable qu'un étudiant puisse récupérer des travaux supprimés par d'autres s'il a recours à un outil de restauration des données effacées.

[4] La liste de ces paramètres n'est pas encore définitive.

[5] Du moins dans un premier temps. Nous estimons qu'il serait préférable de procéder à une livraison incrémentale qui implémenterait d'abord une solution simple, et qui serait éventuellement modifiée pour supporter des heuristiques plus complexes, si cela s'avérait nécessaire.

outils existants en préparant ce qui est nécessaire à chacun et en récupérant les résultats, qui seraient analysés et formatés pour finalement être inclus dans le rapport fourni au correcteur. Les cas suspects seraient clairement identifiés, ce qui permettrait une comparaison manuelle par l'enseignant dans le but de confirmer ou infirmer les « soupçons » soulevés par Oto.

Nous croyons que cette approche permettra l'emploi d'outils concurrents pour renforcer la certitude des résultats du module, ce dernier se contentant de les exécuter et de compiler leurs conclusions. Elle permettrait également de choisir l'outil à utiliser en fonction du langage de programmation cible, ce qui permettrait de profiter des particularités des langages et des forces propres à chacun des logiciels employés à cette fin. Pareille façon de faire apparaît également comme plus simple et rapide à mettre en œuvre, mais intègre également au module les défauts de chacun des outils.

L'ajout de la détection du plagiat comme module Oto utilisable depuis un script semble avantageux à tous les niveaux. Pour le correcteur, il suffira d'ajouter quelques lignes à son script (ce qui ne lui demandera aucune nouvelle connaissance particulière du langage OtoScript) pour bénéficier de cette détection, qui sera ensuite faite de manière complètement automatisée dans le cadre de sa correction de TP de groupe ou individuelle. Une fois le rapport obtenu, il lui suffira de vérifier manuellement les cas douteux signalés par Oto, ce qui est moins lourd qu'une comparaison, deux à deux, de tous les travaux et plus résistant aux tactiques les plus courantes des étudiants.[6]

Mentionnons que pour que la détection du plagiat puisse distinguer sans faute chacun des fichiers formant les projets remis par les étudiants, il sera indispensable pour les travaux remis de respecter strictement une liste de fichiers qui devra être établie à la création de la boîte de remise des TPs, afin de rendre possible leur traitement par le module.[7]

---

[6] D'autant plus que cette attention sera appliquée avec la même rigueur à toutes les copies, ce qui permettra d'éviter que des copies similaires ne soient pas détectées parce qu'elles étaient éloignées l'une de l'autre dans le tas de travaux remis par les étudiants...

[7] Les observations que nous avons faites nous portent à croire qu'en l'absence d'une mesure de validation, les étudiants ont tendance à rendre leurs travaux avec une vaste gamme de noms de fichiers différents. Or, si le module de détection du plagiat s'attend à recevoir un fichier nommé *Darth-Vader.java*, comment ferait-t-il pour savoir qu'il doit le comparer au fichier *le-seigneur-du-mal.java* remis par un étudiant ? Forcer une liste de noms de fichiers, opération que permet déjà Oto, offre aussi l'avantage d'éviter un renommage manuel de tous les fichiers d'une boîte.

B.4 Le choix d'un outil existant

Deux types d'outils de détection de plagiat semblent disponibles, soit ceux ayant recours aux calculs de métriques, et ceux qui procèdent à une analyse lexicale pour transformer le code à analyser en une suite de jetons (37). En raison de la plus grande résistance des seconds envers les tactiques mentionnées plus haut, il semble que nous devrons inclure au moins un tel outil dans notre solution, ce qui explique pourquoi nous nous sommes concentrés sur l'essai de ce genre de logiciels, tel que décrit plus loin.

Pour permettre le développement de notre module de détection du plagiat, nous avons pris en considération plusieurs outils existants et nous en avons fait l'essai sur une série de travaux d'étudiants qui nous ont été fournis par Mélanie Lord, étudiante au doctorat en informatique cognitive et chargée de cours pour le cours INF1120 de l'UQAM à la session d'hiver 2008. Plus précisément, notre corpus de test est fondé sur le travail pratique no. 2 remis par les étudiants, qui consistait au développement, en Java, d'un jeu de bonhomme pendu. Ce programme devait avoir recours à des classes fournies par l'enseignant et le travail devait être fait de façon strictement individuelle.

Dans un article présentant leur propre outil (1), les auteurs du logiciel Plaggie citent un article (37) qui avait comparé les différents logiciels de détection du plagiat, mais en ne retenant que les solutions applicables à Java. Leur choix s'était alors arrêté sur Moss et JPlag, mais, comme ils l'expliquent, ces deux logiciels ne supportent pas une installation locale, ce qui rend moins intéressante leur utilisation dans le contexte d'Oto.[8] Reconnaissant qu'une solution distante n'est pas toujours désirable, l'article suggère comme alternative l'outil YAP3, outil qui, bien que connu, n'est pas adapté à la correction de travaux Java. Étant donné que le support de ce langage est important pour notre contexte, Java étant en ce moment utilisé dans les cours d'introduction à la programmation à l'UQAM, nous devrons l'exclure de nos choix possibles. Le choix d'un outil dépendrait de trois facteurs: la disponibilité du code source (pour nous permettre éventuellement de l'adapter à nos besoins), la possibilité de l'installer localement et

---

[8]Et ce, pour deux raisons majeures. Premièrement, il ne semble pas souhaitable pour l'UQAM de dépendre du service d'une autre institution, qui pourrait décider du jour au lendemain de mettre fin à ce service ou d'en modifier le fonctionnement, ce qui demanderait au mieux un entretien du module Oto. Deuxièmement, avoir recours à une solution extérieure pour le traitement des travaux des étudiants rend difficile le respect du droit d'auteur des étudiants par rapport à leurs travaux, car rien ne nous assure que cette solution extérieure ne conserverait pas une copie du travail qui lui est soumis.

ses capacités de détection.

Dans le but de vérifier la performance des outils répondant aux deux premiers critères, nous avons effectué quelques tests sur quelques-uns des travaux d'étudiants qui nous ont été fournis pour mettre à l'épreuve les outils dont nous avons pu obtenir le code source. Pour ce faire, le travail pratique no. 2 semblait être le meilleur choix pour un tel essai.[9] Les outils que nous avons mis à l'épreuve sont SIM, Plaggie et Sherlock.

SIM (15), qui découpe le fichier en une série de jetons, ce qui le rend peu sensible à des changements superficiels du code source, fut le premier outil que nous avons testé. Il a l'avantage d'être compilable en plusieurs versions visant chacune un langage en particulier. Dans sa version pour Java, il a immédiatement mis en lumière une grande disparité dans la similitude entre les divers travaux, pour un pourcentage aux environs de 0 % pour la plupart jusqu'à un étonnant 84 % pour deux d'entre eux. Les cas à fort pourcentage, considérés suspects, furent inspectés manuellement par nos soins et semblaient effectivement très semblables, avec de légères différences au niveau des noms de variables et du positionnement des accolades, mais avec une structure et des déclarations quasiment identiques. Parmi le groupe, trois travaux nous ont semblé fortement susceptibles d'être en infraction, ne se différenciant les uns des autres que par les modifications esthétiques citées plus haut.

Des résultats similaires à ceux de SIM, mais mettant encore plus en lumière les mêmes cas problématiques, furent produits par Plaggie (1). Ce dernier outil, qui est en quelque sorte une version ouverte de JPlag, fonctionne selon le même principe que SIM, mais retourne encore plus d'informations. Bien qu'ayant l'avantage de supporter la grammaire de Java 1.5, il a comme inconvénient de ne supporter que ce seul langage. Plaggie, qui a donné de bons résultats, s'est toutefois révélé plus sensible que SIM à l'une des faiblesses les plus courantes des outils de détection du plagiat, la détection du *inlining*.

Le logiciel Sherlock (37), quant à lui, a recours à ce que ses auteurs qualifient de *comparaison incrémentielle* entre les différents fichiers, où ces derniers sont comparés plusieurs fois sous forme textuelle avec de moins en moins de caractères que nous supposons moins significatifs (blancs) avant de le faire au moyen de jetons à travers un *parser* qu'ils disent simple et facile à adapter à de nouveaux langages. Malheureusement, Sherlock n'a pas été en mesure

---

[9]Étant donné que ces travaux ne sont composés chacun que d'un seul fichier .java, qu'ils sont plus longs et (en principe) mieux découpés en méthodes qu'un premier TP.

d'identifier les cas de plagiat trouvés par Sim et Plaggie, alors que les heuristiques employées par les présumés plagiaires semblaient naïves. Cependant, il serait nécessaire de soumettre cet outil à davantage de tests avant d'en exclure l'utilisation, l'échantillon utilisé étant de petite taille.

B.5 Les défauts des solutions actuelles

La littérature mentionne plusieurs attaques réussies contre les outils de détection du plagiat. Étant donné que notre solution ferait usage d'outils existants, il semble important d'en être conscient, chaque outil possédant ses forces et ses faiblesses. Déjà dans son document informel (36), notre directeur de recherche faisait mention qu'aucun type d'outil, que ce soit par comptage de métriques ou par une analyse plus poussée du code à vérifier, ne pouvait détecter correctement les cas où l'appel d'une fonction est replacé par son corps. À notre connaissance, aucune piste de solution à ce problème ne semble avoir été explorée, excepté par (4), où les auteurs ont recours à une optimisation de code du compilateur `gcc` qui procédait automatiquement au *inlining* dans les fichiers qui lui étaient soumis. Nous pourrions reprendre cette approche pour la comparaison de programmes C et C++, d'autant plus qu'elle repose sur un outil libre. Si nous souhaitons que notre module puisse aussi résister à l'*inlining* dans le cas d'un programme Java, il serait imaginable de tenter d'utiliser le moteur de *refactoring* de l'éditeur Eclipse pour tenter de procéder au *inlining* de classes Java avant de les comparer.[10] D'autres cas où la détection du plagiat sera généralement moins efficace sont détaillés par Prechelt, Malphol et Philippsen (26), qui présentent des dizaines de cas et qui décrivent comment ces cas furent efficaces pour contourner l'outil JPlag. Par exemple, l'un de ces cas serait l'ajout d'instructions inutiles dans le corps des procédures d'un programme (déclaration de variables, initialisations, instanciations), contre lequel, à notre connaissance, les logiciels actuels demeurent impuissants. Cela dit, si nous décidons de tenter de combler les défauts des outils pris individuellement, il serait intéressant de mettre au point une suite de tests permettant de juger de la qualité d'un outil de détection du plagiat, permettant ainsi d'avoir une idée de la résistance de nos outils face aux divers cas possibles, ce qui n'est pas faisable avec un simple échantillon de travaux d'étudiants qui, bien que s'avérant de vrais cas, ne prétendent pas couvrir toutes les attaques possibles.

---

[10] À moins que nous tentions d'avoir recours à la même astuce en compilant les programmes des étudiants avec GCJ auquel nous aurions spécifié un niveau d'optimisation qui inclurait le *inlining*.

B.6 Conclusion

Face à la facilité avec laquelle il est possible de copier un travail de programmation ainsi que l'existence de nombreuses astuces permettant de masquer la ressemblance entre deux programmes (avec un succès variable), combinées à des groupes d'étudiants qui sont souvent de grande taille, l'automatisation partielle de la détection du plagiat apparaît comme une solution naturelle, étant une suite logique à l'automatisation partielle de la correction des travaux. Notre approche, qui consisterait en l'emploi d'outils externes spécialisés chacun en une tâche précise, simplifie le développement d'un module voué à la détection de la triche et permet une plus grande modularité tout en en facilitant la maintenance future. Elle nous permettera aussi de limiter les faiblesses de chacun des outils pris individuellement, au lieu de réinventer la roue en créant de toute pièce un nouvel outil de détection de plagiat.

APPENDICE C

ANALYSE DE L'APPLICATION WEB D'OTO

Dans cet appendice, nous traiterons de l'application Web d'Oto (31). Cet exercice se veut un complément à notre analyse d'Oto qui était présentée dans ce même chapitre. Nous considérerons l'application Web sous plusieurs aspects. D'abord, nous discuterons de son architecture et de sa mise en œuvre interne. Nous présenterons ensuite son interface du point de vue de l'utilisabilité. Nous détaillerons certaines modifications mineures que nous y avons effectué avant de présenter nos conclusions et idées de projets liés à l'application Web.

C.1 Mise en œuvre

Dans cette section, nous traiterons de la mise en œuvre de l'application Web au niveau des choix technologiques et de sa constructions. Nous n'aborderons pas la question de la qualité des interfaces utilisateur, qui sera l'objet de la section C.2.

C.1.1 Technologies utilisées

À l'interne, l'application Web est un mélange de *Java Server Pages* (JSP) et d'un *servlet* Java. Les premières sont chargées d'effectuer le fort du travail, la dernière ne servant qu'à rediriger l'utilisateur vers la page d'accueil. Du côté de l'utilisateur, le code employé est, bien entendu, du HTML. Le JavaScript a également été utilisé pour dynamiser le comportement des pages du côté client, ce qui permet de soulager la charge sur le serveur Web.

L'application elle-même ne contient pas une copie locale d'Oto, ni même une partie de son code source. Elle manipule Oto à travers ses commandes sur le *bash* Unix. Pour ce faire, l'application Web doit établir une connexion SSH avec le serveur Arabica. Cette connexion se

fait dans le contexte de l'utilisateur. Celui-ci doit d'abord s'authentifier avec son code d'usager du laboratoire Unix de l'UQAM. Un *bean* Java s'occupe d'établir la connexion. Un *bean* est une classe Java respectant certaines conventions dans sa structure, devant notamment posséder un constructeur par défaut, ses accesseurs et mutateurs devant respecter une convention de nommage et la classe elle-même devant « implémenter » l'interface *Serializable*. Celui-ci est conservé durant un temps défini par les réglages du serveur Web, ou jusqu'à ce que l'utilisateur sélectionne l'option lui permettant de se déconnecter.

Le logiciel du serveur exécutant l'application est Apache Tomcat 5.5.9, un logiciel libre.

C.1.2 Limitations et critiques de la mise en œuvre

Au niveau de la construction de l'application, trois critiques principales peuvent être faites.

D'abord, la qualité du code source est faible. Les fichiers JSP sont mal indentés et les parenthèses sont souvent mal placées et incorrectement alignées (ce qui complique leur appariement, d'autant plus que le code Java est placé entre des balises le différenciant du code HTML). De nombreux fichiers contiennent du code dupliqué, particulièrement au niveau du JavaScript, alors qu'il aurait été préférable de le placer dans des fichiers importables (*.js*). De plus, les pages HTML ne respectent pas le standard 4.01 énoncé par le *World Wide Web Consortium*, alors que leurs entêtes affirment le contraire. Cette situation est d'autant plus incompréhensible qu'il s'agit d'une ancienne norme relativement peu contraignante. En effet, selon Wikipédia (`http://www.fr.wikipedia.org/wiki/Hypertext_Markup_Language`), elle a été adoptée en décembre 1999, soit environ sept ans avant le développement de l'application Web d'Oto.

Ensuite, un code de mauvaise qualité engendre une faible maintenabilité, et ce pour deux raisons. Premièrement, la mauvaise disposition du code rend difficile la localisation du code à modifier lorsqu'il devient nécessaire d'y apporter des changements (par exemple, lorsque nous avons modifié les paramètres de la commande `creer_boite`). Deuxièmement, l'absence de factorisation du code partagé par les différentes pages de l'application rend la modification de l'ensemble problématique, car il sera nécessaire de copier tout changement à l'ensemble des pages contenant le code ayant été modifié, au risque d'en oublier ou de briser le fonctionnement du code en plage.

Pour faire suite au problème de la maintenabilité, l'application souffre de limitations importantes au niveau de la testabilité. Elle ne contient, en effet, aucun test automatisé. Cette absence se fait sentir tant au niveau des éléments de l'interface des pages que du code derrière. Même si, au cours de nos travaux, nous avons repéré plusieurs problèmes dans l'application, notamment au niveau de la validation des formulaires, nous hésitions à tenter de les corriger en l'absence de tests de non-régression. Cette situation est d'autant plus irritante que plusieurs cadres de tests existants permettent de faciliter les tests des applications Web. Une rapide recherche nous a permis de découvrir IeUnit (`http://code.google.com/p/ieunit/`), un cadre de tests unitaires pour les pages Web, JsUnit (`http://jsunit.net/`), une adaptation de JUnit pour le langage JavaScript et JTF (`http://jtf.ploki.info/`), destiné à tester le fonctionnement d'un site avec plusieurs fureteurs. Il est fort probable qu'une recherche plus exhaustive nous aurait permis d'en identifier encore davantage.

C.2 Interface utilisateur et utilisabilité

Au-delà de la mise en œuvre de son code source et des qualités non-fonctionnelles de celui-ci, l'interface utilisateur de l'application Web souffre de plusieurs défauts importants, notamment au niveau de son apparence peu soignée, mais surtout de sa faible utilisabilité. Dans cette section, nous traiterons de la question de l'utilisabilité des interfaces de l'application. Nous présenterons d'abord cette qualité non-fonctionnelle avant d'effectuer une analyse d'un sous-ensemble de pages de l'application à la lumière de celle-ci. Les résultats que nous obtiendrons nous aiderons à établir des conclusions et des idées de travaux futurs reliés à cette application.

C.2.1 Définition

Selon la norme ISO 9241-11,

> *Un système est utilisable lorsqu'il permet à l'utilisateur de réaliser sa tâche avec efficacité, efficience et satisfaction dans le contexte d'utilisation spécifié.*[1]

Dans un tel contexte, l'efficacité signifie que la tâche est accomplie correctement, l'efficience signifie que les ressources utilisées pour accomplir la tâche (le temps et l'effort requis) sont

[1] `http://www.usabilis.com/methode/test-utilisateur.htm`

raisonnables dans le contexte d'utilisation et la satisfaction est une mesure subjective de l'attitude de l'utilisateur durant la réalisation de la tâche.

C.2.2 Utilisabilité Web

Dans le cas d'une application Web, le problème de l'utilisabilité est d'autant plus important que l'utilisateur ne disposera que rarement d'une formation spécifique à cette application particulière. Cette situation rend difficile la réalisation par le développeur d'une solution prenant en compte les caractéristiques physiologiques de toutes les catégories d'utilisateurs potentiels de son application.[2]

Face à cette difficulté, des chercheurs comme Jakob Nielsen ont établi des heuristiques que peuvent utiliser les développeurs pour éviter les erreurs les plus courantes en terme d'ergonomie logicielle. C'est en nous basant sur ces heuristiques que nous analyserons l'interface de l'application Web d'Oto pour mettre en évidence ses défauts. L'exercice que nous avons effectué ici, une analyse heuristique, est l'une des techniques permettant de découvrir les défauts des interfaces, l'autre technique majeure étant le test d'utilisabilité. Le reste de cette section ne fournira que les informations les plus importantes, car l'utilisabilité des logiciels, bien qu'étant un sujet intéressant, n'est pas le sujet principal de ce mémoire. Pour la même raison, nous avons évité de traiter ici d'un autre sujet en émergence, l'accessibilité des sites Web aux gens présentant des déficiences physiologiques et le support des fureteurs vocaux ou à affichage braille.

C.2.3 Heuristiques de Nielsen

Les heuristiques de Nielsen, qu'il a présentées notamment dans un article publié en 1990 (25), sont des pratiques recommandées au niveau des interfaces, développées à partir d'observations empiriques. Dans cette section, nous les présenterons avec, pour chacune, une courte explication.

1. *Visibilité de l'état du système* : À tout moment, l'utilisateur doit être en mesure de savoir

---

[2] Combien de fois avons-nous entendu parler de la mauvaise réputation des développeurs d'applications au niveau de la conception d'interfaces... l'auteur de ces lignes, à se sujet, se donne le droit de citer Philippe Gabrini, professeur d'informatique à l'UQAM, qui lui a répété à plusieurs reprises : « Nous, les programmeurs, nous sommes nuls en interfaces… »

dans quel état se trouve le système. Un site de vente en ligne affichera, par exemple, l'étape en cours du processus de commande ainsi que le nombre et la nature des étapes passées et à venir.

2. *Adéquation du système au monde réel* : Les termes utilisés dans le système doivent être compréhensibles par l'utilisateur. Ils doivent respecter le vocabulaire courant de celui-ci, même si cela peut être différent d'un terme technique plus exact (par exemple, billet de métro au lieu de titre de transport).

3. *Contrôle et liberté de l'utilisateur* : L'utilisateur doit pouvoir se sortir lui-même de situations d'erreurs, notamment lorsqu'il a choisi la mauvaise option.

4. *Cohérence et standards* : L'interface doit être cohérente au niveau de sa présentation, des termes utilisés et de son comportement. Si des standards existent dans le domaine pour lequel l'interface est construite, ces standards doivent être respectés.

5. *Design minimaliste* : L'interface ne doit pas contenir d'éléments visuels superflus. Elle doit contenir le minimum d'informations nécessaire pour accomplir la tâche. Une autre description de ce principe serait de « minimiser l'encre ». Il s'agit de considérer l'interface comme une feuille de papier sur laquelle il faut afficher le nécessaire en utilisant le moins d'encre possible.

6. *Reconnaissance plutot que rappel* : La mémoire de travail d'un individu étant limitée à 7 ± 2 éléments (24), il faut autant que possible rappeler à l'utilisateur les éléments nécessaires au traitement d'une étape de la tâche au lieu de se fier à sa mémoire.

7. *Flexibilité d'utilisation* : Autant que possible, l'interface doit être utilisable au moyen de plusieurs dispositifs : souris, clavier, etc. Les raccourcis clavier doivent permettre l'utilisation plus rapide de l'interface.

8. *Aide à la gestion des erreurs* : Les erreurs survenant à l'exécution doivent être mentionnées à l'utilisateur dans une langue claire et de manière à ce qu'il puisse comprendre. Un exemple de ce qu'il ne faut pas faire serait une erreur telle que « Erreur dans le VXD à l'adresse 0EFF:3A47. »

9. *Prévention des erreurs* : Les interfaces doivent être conçues de manière à prévenir les erreurs. Par exemple, un modèle peut être inclus dans un champ texte destiné à recevoir un numéro de téléphone.

10. *Aide et documentation* : Pour chaque fonctionnalité de l'interface, une aide en ligne doit être disponible.

C.2.4 Description de pages types de l'application Web

Dans cette section, nous considérerons en détails deux des interfaces de l'application Web. Pour chacune, nous la décrirons en mentionnant les défauts et problèmes rencontrés.

C.2.4.1 Écran de connexion

D'abord, analysons l'écran de connexion, qui est la première page atteinte par l'utilisateur de l'application Web d'Oto. Un logo occupe le centre du haut de l'écran, qui sert à identifier l'application comme étant Oto. En dessous, un formulaire permet à l'utilisateur de saisir son code d'accès et son mot de passe du laboratoire Unix de l'UQAM et de choisir s'il désire employer l'application à titre d'étudiant ou d'enseignant, qui n'ont pas accès aux mêmes fonctionnalités. La confirmation de la connexion se fait à l'aide d'un bouton identifié « Se connecter ». Sous le formulaire, nous pouvons trouver des informations concernant certains collaborateurs au projet Oto, ainsi que le logo stylisé de l'UQAM.

Le premier reproche que nous pouvons addresser à cet écran, bien qu'il ne soit pas visible sur la figure pour économiser de l'espace, est le titre de la page, « Connexion », qui ne fait pas référence à Oto. Dans le corps de la page, un défaut est la perte d'espace causée par le logo, qui est déployé entièrement à la verticale, en gaspillant de l'espace horizontal. De plus, cet identificateur, bien qu'étant formé de texte, est mis en œuvre sous la forme d'une image, ce qui l'empêche de s'adapter à un éventuel redimensionnement de la fenêtre du fureteur et peut devenir trop petit pour être lisible si la résolution de l'écran était augmentée. Un autre problème au niveau de l'image est l'absence, dans la balise HTML qui en permet l'affichage, de la valeur *alt*, qui est destinée à afficher un message texte au cas où l'image ne pouvant être affichée, comme ce serait le cas d'un fureteur en mode texte. Ensuite, le formulaire est, lui aussi, mal conçu. En premier lieu, son fond de couleur alourdit l'interface et rend difficile la lecture du texte écrit au-dessus. Au niveau du texte, la taille du caractère utilisé est trop petite, ce qui rend difficile sa lecture, d'autant plus que l'interface dispose de suffisamment d'espace pour employer un caractère plus gros. Un autre problème est le positionnement des boutons radio permettant de choisir entre une connexion en tant qu'étudiant ou en tant qu'enseignant.



Figure C.1 L'écran de connexion de l'application Web.

Vraisemblablement, le concepteur de l'interface était convaincu que la majorité des utilisateurs de cette interface seraient des étudiants, car cette valeur est choisie par défaut. Or, la valeur « Enseignant » est mise plus en évidence que la valeur « Etudiant », étant à gauche, ce qui est contradictoire. De plus, leur positionnement à l'horizontal, aligné sur les étiquettes à la gauche brise le déplacement de l'oeil qui doit se déplacer en zig-zag au lieu de sélectionner les options les unes après les autres. Finalement, les boutons ne sont pas placés de manière optimale, car le bouton « Se connecter » n'est pas aligné avec les champs texte, ce qui rend sa localisation plus difficile. Par ailleurs, le bouton « Annuler », qui vide les champs texte, nous semble inutile, car le formulaire est de petite taille, d'autant plus que cette page est un écran de connexion où rien n'a été fait (et, par conséquent, rien n'est à « annuler ».)

C.2.4.2 Écran de vérification d'un TP

Considérons maintenant un écran que les étudiants pourraient être amenés à utiliser à l'occasion, soit celui leur permettant de faire une vérification de leur TP. La forme de la page, son apparence, le positionnement du menu et la conception des *Widgets* de la plupart des écrans sont très similaires à celle-ci.

L'un des aspects positifs de cet écran est que contrairement à la connexion, l'écran de vérification d'un TP occupe tout l'espace horizontal disponible. Par contre, la taille des éléments affichés est mise en œuvre avec des valeurs absolues, ce qui ne permet pas de redimensionnement correct, alors qu'il aurait été souhaitable de pouvoir redimensionner la fenêtre et que les éléments de l'écran s'y adaptent, ce qui est appelé du « design liquide ». Cette page est divisée en quatre sections distinctes.

Une bande en haut et à gauche de l'écran semble indiquer que l'utilisateur est authentifié en tant qu'étudiant : « OTO_ETUDIANT ». Si ce rappel n'est pas mauvais en tant que tel, le choix d'un identificateur blanc sur un fond pâle en rend la lecture difficile. De plus, cette expression n'est pas aussi claire qu'elle aurait pu l'être, car il serait possible d'en déduire à tort qu'Oto est un étudiant, ou même que l'utilisateur est authentifié en tant qu'« Oto ».

À droite de la bande dont nous venons de traiter se trouve une autre bande de couleur, qui comporte deux références textuelles. La première, à gauche, est nommée « Guide d'utilisation » et est, comme son nom l'indique, un lien vers un tel guide en format pdf. La seconde référence textuelle, à la droite, sert à indiquer sous quel nom l'utilisateur est connecté : « Usager =

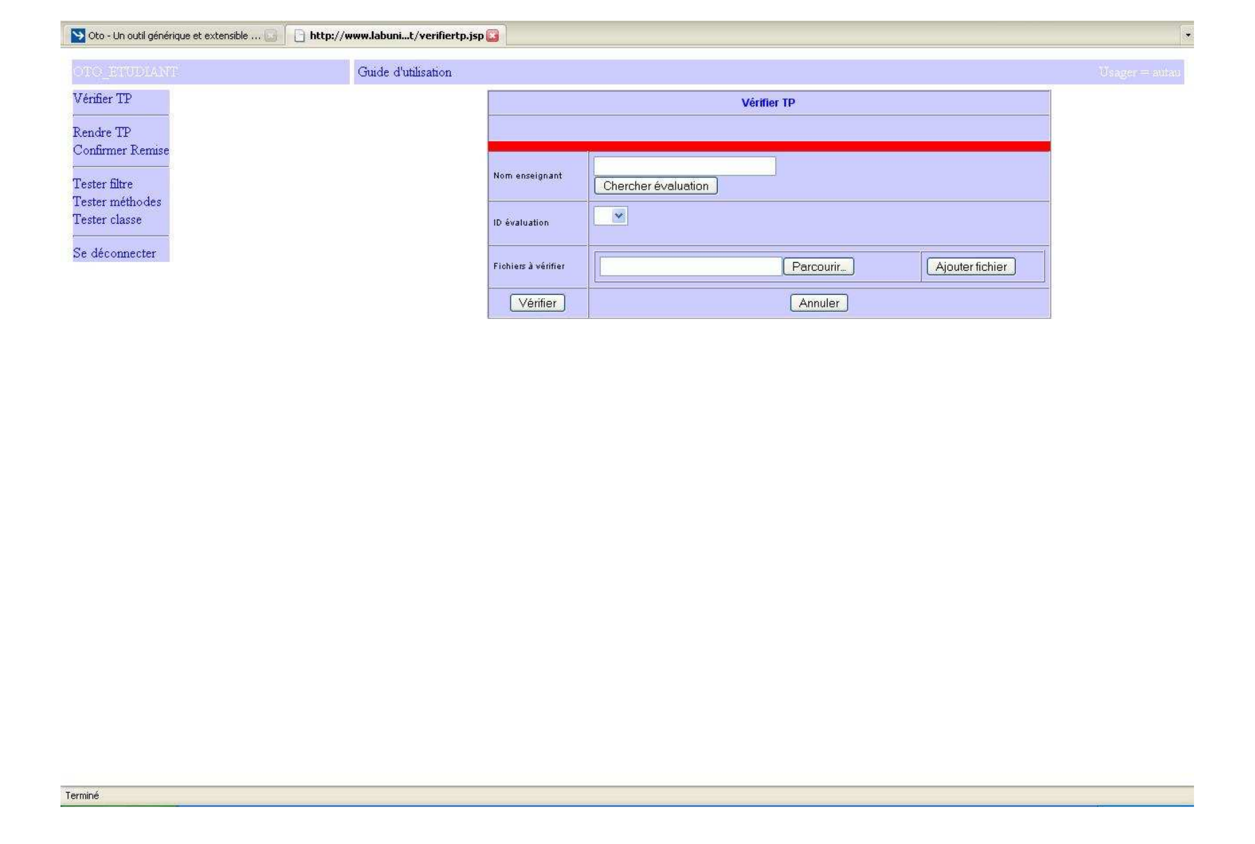


Figure C.2 L'écran de vérification d'un TP, connecté en tant qu'étudiant.

le_nom ». L'idée de fournir un lien vers une fonctionnalité d'aide à l'utilisateur nous plaît à prime abord. Cependant, l'emploi d'un fichier au format pdf à cette fin n'est pas idéal, et ce pour deux raisons, la première étant que les logiciels permettant de lire un tel fichier ne sont pas forcément installés sur l'ordinateur de l'utilisateur, rendant caduque l'offre d'aide dans un tel cas et la seconde étant que même si l'ouverture du fichier est un succès, celui-ci sera ouvert à la première page, et non celle traitant de la fonctionnalité en cours, demandant un effort supplémentaire à l'utilisateur qui pourrait être tenté d'abandonner simplement son utilisation. Une meilleure approche aurait été de développer une véritable fonctionnalité d'aide intégrée à l'interface, permettant la recherche et pouvant se référer directement à la fonctionnalité en cours d'utilisation.

Une troisième bande est placée à la verticale de la gauche de l'écran. Elle contient des liens vers les différentes fonctionnalités offertes à l'utilisateur-étudiant, en plus de la possibilité de quitter l'application en se déconnectant. À son propos, nous ferons trois remarques. En premier lieu, ses boutons de format texte bleu foncé sur un fond magenta sont difficiles à lire. Nous apprécions par contre le fait que placer le curseur de la souris par dessus change la couleur du fond pour l'orange, ce qui met la sélection en évidence. Toutefois, la zone où les boutons sont effectifs ne couvrent que le texte et n'exploite pas tout l'espace disponible, toute la largeur du menu, ce qui aurait été préférable en augmentant la marge de manœuvre de l'utilisateur pour cliquer au bon endroit. Troisièmement, nous estimons que le positionnement des boutons n'est pas optimal, la déconnexion étant présentée comme une fonctionnalité alors que cela n'est pas le cas. Il eut été préférable de positionner le bouton de déconnexion à proximité de l'indicateur identifiant l'utilisateur connecté, regroupant ainsi leurs fonctions dans la page.

La dernière zone de l'écran est spécifique à la fonctionnalité choisie par l'utilisateur, dans ce cas la vérification d'un TP. Elle est formée par un tableau HTML qui inclut un formulaire. Sa partie supérieure contient une étiquette indiquant à l'utilisateur quelle est la fonctionnalité en cours. Elle est rendue nécessaire par le fait que le menu ne désactive pas le bouton qui lui correspond. En-dessous se trouve une bande rouge, qui n'est populée qu'en cas de problème avec les paramètres fournis plus bas. Sous celle-ci se trouve un champ texte qui permet de saisir le nom de l'enseignant ayant créé l'évaluation sur laquelle l'utilisateur veut vérifier son travail. Ce dernier doit ensuite cliquer sur le bouton « Chercher évaluation » pour que les évaluations associées à cet enseignant soient récupérées et aillent populer une liste déroulante située en dessous et identifiée « ID évaluation ». Ce dernier terme est incohérent avec le vocabulaire

d'Oto où l'identification porte un « nom ».

Sous l'identification de l'évaluation, la liste des fichiers à téléverser à l'application doit être spécifiée. Celle-ci se fait au moyen d'un champ texte et d'un bouton ouvrant une boîte de choix de fichier. Une fois le fichier spécifié, il est possible d'ajouter d'autres fichiers au moyen du bouton « Ajouter fichier » à la droite du champ. Cette section du formulaire contient une dissonance cognitive, car le bouton ne confirme pas l'ajout du fichier à l'évaluation, mais ajoute un autre champ.

La vérification se fait au moyen d'un bouton « Vérifier » situé en base du formulaire. Toutefois, le retour charriot du clavier enclenche le processus de vérification à tout moment dans l'utilisation du formulaire. Un tel comportement entraîne fréquemment des erreurs à l'utilisation.

C.2.5 Résultats

L'application Web d'Oto est peu utilisable. Ses pages ne respectent pas plusieurs des heuristiques de Nielsen. Le tableau C.1 contient, pour chacune des pages, les heuristiques n'ayant pas été respectées. En plus des pages que nous avons vues précédemment, le tableau contient également trois autres pages parmi celles les plus suceptibles d'être utilisées par les utilisateurs de l'application. Les résultats sont sans équivoques, les interfaces ne respectant pas une majorité des heuristiques.

Le choix de couleurs des interfaces, combiné à un mauvais contraste, rend difficile la lecture du texte et la navigation entre les pages. L'espace disponible n'est pas bien exploité, l'application ne gérant pas bien les positionnements relatifs en langage HTML. De plus, plusieurs erreurs dans le code JavaScript provoquaient des problèmes d'affichage avec le fureteur Mozilla Firefox jusqu'à la version 3. Ces erreurs étaient provoquées le code qui manipule les éléments des pages par programmation. Un autre problème est que les erreurs à l'utilisation sont signalées par un champ rouge vif devenant vite agressant, dans lequel est écrit un texte pâle et mince, difficile à lire. Ces messages d'erreurs sont souvent mal placés et se contentent d'afficher à l'écran l'erreur telle que l'interface texte d'Oto la fournit sans formatage. De manière générale, l'application Web possède une apparence peu soignée et peu professionnelle.

Heuristique non respectée	**Accueil**	**Correction d'un groupe**	**Création d'une boîte**	**Remise d'un TP**	**Vérification d'un TP**
Visibilité de l'état du système			x		x
Adéquation du système au monde réel	x			x	
Contrôle et liberté de l'utilisateur	x	x	x	x	x
Cohérence et standards	x	x	x	x	x
Design minimaliste	x	x	x	x	x
Reconnaissance plutot que rappel		x			
Flexibilité d'utilisation		x	x	x	x
Aide à la gestion des erreurs	x				
Prévention des erreurs		x		x	x
Aide et documentation	x	x	x	x	x

Tableau C.1 Heuristiques non respectées par les pages de l'application Web.

C.3 Conclusion et travaux futurs

L'application Web d'Oto comporte des défauts, tant dans sa mise en œuvre que de la conception de ses interfaces utilisateur. Leur correction demanderait un effort considérable. Une simple modification de la page d'accueil que nous avons effectuée a demandé plusieurs heures de travail, alors qu'il s'agissait d'un déplacement et d'un renommage d'éléments. Modifier les autres interfaces, lesquelles étant d'une complexité bien plus grande, exigerait vraisemblablement beaucoup plus de temps. Vue la qualité de l'application en général, une telle correction semble peu intéressante. Cette situation est d'autant plus problématique que la présence d'une application Web peut s'avérer un atout fort intéressant pour l'utilisation et la diffusion d'Oto. Repartir sur de meilleures bases apparaît comme étant une meilleure solution.

Face à cette situation, nous n'avons d'autre choix que de recommander le développement d'une nouvelle interface Web pour Oto. Celle-ci pourrait viser le développement d'une application Web se concentrant sur les qualités de testabilité et d'utilisabilité. Il pourrait s'agir d'un projet de recherche intéressant pour une maîtrise en informatique ou en génie logiciel. Notons qu'au moment de rédiger ce mémoire le développement d'une nouvelle interface Web était en projet.

La manipulation de cette nouvelle interface pourrait certainement être facilitée par le mise en œuvre d'une extension pour Mozilla Firefox. Elle permettrait d'intégrer l'utilisation de l'application Web dans le fureteur lui-même, évitant aux étudiants de devoir connaître son adresse et accélérant l'accès aux options les plus suceptibles d'être utilisées par eux. Cette extension pourrait être installée sur les postes de travail du laboratoire des sciences de l'UQAM où les laboratoires pratiques de programmation sont tenus. Elle pourrait également être téléchargeable depuis le site des enseignants des cours de programmation pour que les étudiants puissent l'utiliser depuis leurs propres ordinateurs. Le développement d'une telle extension représenterait sans doute un défi intéressant pour occuper, du moins partiellement, un stage d'initiation à la recherche pour un étudiant de premier cycle.

APPENDICE D

RÉSULTATS DU PROFILAGE D'OTO 1+

Dans cet appendice, nous présenterons les résultats détaillés du profilage de différents cas d'utilisation d'Oto 1+. L'intérêt de telles analyses était de vérifier la théorie selon laquelle Oto passait une majorité du temps de ses corrections à utiliser des programmes externes. Une telle validation était importante pour décider des modifications à entreprendre par la suite, particulièrement au niveau de l'amélioration des performances de l'outil. Pour ne pas alourdir inutilement cet appendice, nous présenterons deux cas représentatifs des résultats que nous avons obtenus, soit l'ensemble des tests intégrés d'Oto, et une correction de groupe utilisant le module de détection du plagiat.

Ces analyses ont été faites au moyen de l'outil de profilage *ruby-prof*. Celui-ci dispose de deux modes de fonctionnement. D'abord, la manière la plus simple de l'utiliser consiste à avoir recours à son exécutable pour exécuter un programme Ruby en lieu et place de l'interprète Ruby standard (auquel il aura lui-même recours à l'interne). De plus, il est également possible d'utiliser *ruby-prof* directement dans le code source en démarrant et en arrêtant explicitement le profileur autour du code à tester. Pour éviter d'altérer le fonctionnement d'Oto et de ses modules, nous avons choisi la première approche.

D.1 Suite de tests complète

La suite de tests complète d'Oto a pour but de vérifier le bon fonctionnement de l'ensemble de l'application. Pour atteindre ce but, elle comprend une série de fichiers définissant des tests adaptés à la nature variable des composants vérifiés : appels d'une autre instance d'Oto au niveau des commandes, tests réflexifs pour le moteur, etc.

Les extraits les plus importants des résultats obtenus suite à l'exécution de *ruby-prof* sont présentés à la figure D.1. Ceux-ci étaient largement plus volumineux que ce que nous pouvons inclure dans l'espace limité de la figure, mais contenaient essentiellement un grand nombre de *threads* de courte durée. Nous nous concentrons sur le *thread* principal de l'application, et plus particulièrement sur le pourcentage de temps que celui-ci consacre à ses différentes tâches. Les temps indiqués sur la figure, à l'exception des pourcentages, sont en secondes.

Les résultats obtenus sont intéressants. Les trois méthodes en pourcentage de temps utilisé (les trois premières lignes du résultat) sont tous des appels systèmes, c'est à dire des appels de programmes externes à Oto par l'interpréteur Ruby. Ensemble, ils occupent 63,01 % du temps de l'exécution, et ce, malgré le fait que ces méthodes ne soient invoquées que 407 fois sur les dizaines de milliers de méthodes invoquées pour l'ensemble de l'exécution. Les autres méthodes, essentiellement consacrées à la manipulation de répertoires et d'entrées/sorties, sont moins dominantes en terme de pourcentage. Il est à noter que si Oto passe une quantité non négligeable de temps (15,52 secondes) à exécuter la méthode `each` sur des tableaux, il s'agit de code exécuté à l'intérieur de blocs de code qui n'influence pas négativement les temps d'exécution, comme le montre la valeur « self ».[1]

D.2 Correction d'un groupe (détection du plagiat)

À la figure D.2 se trouvent les extraits les plus importants du profilage d'une correction de groupe réalisée par Oto. Celle-ci, qui a été réalisée sur les 37 TP d'un groupe d'étudiants, consiste en un appel conventionnel du module de détection du plagiat. Ce dernier a comme particularité d'effectuer un nombre élevé de manipulation de dossiers et de fichiers étudiants, en plus d'avoir recours au logiciel externe pour la comparaison des travaux.

Dans l'extrait présenté à la figure, nous constatons qu'Oto passe le plus clair de son temps (47,73 %) à effectuer des appels systèmes, consacrés majoritairement dans ce cas précis au logiciel SIM (voir la section 1.4.1). Les autres méthodes concernent essentiellement la manipulation de fichiers, notamment au niveau du chargement de la classe de mise en œuvre de la comparaison (qui est chargée depuis le module) et de listes.

[1] Rendant de ce fait illusoire le nombre élevé de secondes associées à ces méthodes.

%self	total	self	wait	child	calls	name
43.69	11.93	9.93	0.00	2.00	195	<Module::Timeout>#timeout
13.51	3.10	3.07	0.00	0.03	130	Kernel#‘
5.81	1.34	1.32	0.02	0.00	82	Kernel#system
3.78	0.86	0.86	0.00	0.00	2172	IO#read
2.60	0.59	0.59	0.00	0.00	6957	<Class::File>#old_expand_path
2.55	0.92	0.58	0.00	0.34	1234	Kernel#require
1.89	0.43	0.43	0.00	0.00	561	Time#initialize
0.84	0.20	0.19	0.00	0.01	1821	<Class::File>#stat
0.70	0.16	0.16	0.00	0.00	3	Kernel#sleep
0.70	0.16	0.16	0.00	0.00	14947	<Class::File>#join
0.66	0.33	0.15	0.00	0.18	400	Kernel#require-1
0.66	0.15	0.15	0.00	0.00	1445	<Class::Dir>#open
0.62	0.36	0.14	0.00	0.22	63	Kernel#load
0.62	0.22	0.14	0.00	0.08	2513	<Module::OsUtils>#separer
0.53	0.12	0.12	0.00	0.00	4125	String#split
0.53	0.12	0.12	0.00	0.00	313	<Class::IO>#read
0.53	0.12	0.12	0.00	0.00	143	IO#flush
0.48	0.11	0.11	0.00	0.00	3328	File#initialize
0.44	15.52	0.10	0.00	15.42	5999	Array#each-4
0.40	0.09	0.09	0.00	0.00	560	<Class::Dir>#old_bracket
0.40	0.09	0.09	0.00	0.00	169	Module#include
0.40	3.95	0.09	0.00	3.86	371	<Module::Oto>#noop
0.35	0.08	0.08	0.00	0.00	14314	<Class::Object>#allocate
0.31	0.08	0.07	0.00	0.01	1245	Array#reject
0.31	0.07	0.07	0.00	0.00	1517	IO#write
0.31	0.24	0.07	0.17	0.00	195	<Class::Thread>#start
0.31	0.07	0.07	0.00	0.00	904	<Class::Dir>#rmdir

Figure D.1 Sortie du profilage de la suite de tests d'Oto.

%self	total	self	wait	child	calls	name
47.73	3.58	3.57	0.00	0.01	1589	Kernel#system
19.12	1.53	1.43	0.00	0.10	4767	<Class::File>#utime
3.07	0.25	0.23	0.00	0.02	37	Kernel#load-1
1.87	0.15	0.14	0.00	0.01	8472	<Module::Pass>#if
1.47	3.71	0.11	0.00	3.60	1552	Object#nettoyerApres-4
0.94	0.09	0.07	0.00	0.02	4960	SystemCallError#initialize
0.80	0.06	0.06	0.00	0.00	5043	File#initialize
0.80	0.06	0.06	0.00	0.00	37	IO#gets
0.80	5.94	0.06	0.00	5.88	1825	Array#each-1
0.80	0.07	0.06	0.01	0.00	148	Thread#kill
0.67	0.06	0.05	0.00	0.01	154	<Class::Struct>#new
0.67	0.11	0.05	0.00	0.06	71	Kernel#require-4
0.67	0.05	0.05	0.00	0.00	500	IO#read
0.53	0.04	0.04	0.00	0.00	5043	IO#close
0.53	1.67	0.04	0.00	1.63	23280	Array#each-3
0.53	0.07	0.04	0.00	0.03	888	<Module::OsUtils>#separer
0.53	6.77	0.04	0.00	6.73	2353	Array#each
0.53	0.05	0.04	0.00	0.01	1553	Dir#each
0.53	0.26	0.04	0.00	0.22	4961	<Class::IO>#open
0.53	0.04	0.04	0.00	0.00	1553	<Class::Dir>#open
0.40	0.03	0.03	0.00	0.00	15840	String#==
0.40	0.16	0.03	0.00	0.13	4843	Object#assertNettoyable
0.40	0.06	0.03	0.00	0.03	58	Kernel#require-5
0.40	0.17	0.03	0.00	0.14	4843	String#nettoyer
0.40	1.75	0.03	0.00	1.72	4767	<Module::FileUtils>#touch
0.40	0.03	0.03	0.00	0.00	9921	String#split
0.40	7.19	0.03	0.00	7.16	74	Object#nettoyerApres-1

Figure D.2 Sortie du profilage d'une correction de groupe (détection du plagiat).

D.3 Synthèse et analyse des résultats obtenus

Les résultats que nous avons obtenus confirment notre hypothèse selon laquelle Oto consacre une majorité de son temps à exécuter des programmes externes. À la lumière de cette conclusion, l'amélioration d'Oto passera vraisemblablement par deux approches distinctes. La première sera d'améliorer tout ce qui concerne l'encadrement de l'exécution des scripts ainsi que les scripts eux-mêmes, afin de diminuer l'estimation asymptotique du temps et du travail nécessaire à l'exécution des scripts lorsque cela est possible, particulièrement pour les corrections intra-groupe. En d'autres mots, Oto devra réduire le surcoût associé à l'exécution des tâches de correction. De plus, il sera nécessaire possible d'optimiser le fonctionnement des modules pour minimiser le temps nécessaire à l'exécution de programmes externes.

APPENDICE E

SYNTAXE DES SCRIPTS DU DSL OTO

Dans cet appendice, nous examinerons la syntaxe du DSL Oto, particulièrement au niveau des règles à respecter pour accéder aux diverses fonctionnalités de l'outil. À la base, le code source des scripts Oto doit être un programme Ruby valide et respectant la syntaxe établie par Ruby.

Les diverses possibilités d'utilisation du DSL Oto seront présentées ici sous forme d'exemples, dont la légende est présentée au tableau E.1. Lorsque cela sera nécessaire, nous préciserons textuellement certains détails concernant la syntaxe à utiliser et les possibilités du DSL.

Élément	**Description**
Barre verticale (|)	Éléments exclusifs entre eux (OU logique exclusif).
Pointillés (. . .)	Possibilité d'avoir recours à d'autres paramètres similaires.
Texte en italique (*italique*)	Les éléments en italique doivent être remplacés par les valeurs appropriées.
Texte entre crochets ([texte])	Les éléments entre les crochets sont optionnels. Les crochets eux-mêmes doivent être omis (sauf dans le cas de la conservation des résultats et des paramètres des rapports).
Texte simple	Nom à utiliser tel quel à l'utilisation.

Tableau E.1 Légende des exemples de la syntaxe d'Oto.

E.1 Appel à Oto sur la ligne de commande

E.1.1 Exécution de commandes Oto

```
oto nom_commande [paramètres] [--parametres={param1=valeur1,...}] [fichiers] [*tp_oto]
```

E.1.2 Exécution par fichier *.oto*

```
oto script.oto [--parametres={param1=valeur1,...}] [fichiers] [*tp_oto]
```

E.2 Environnement d'exécution

E.2.1 Méthodes disponibles dans le script

Le tableau E.2 présente un certain nombre de méthodes offertes par la classe `EnvironnementExecution` et faisant partie de l'interface de programmation des scripts. Ces méthodes donnent des informations sur le groupe à corriger et le script de correction utilisé. Plus particulièrement, les méthodes de l'objet `groupe` sont présentées au tableau E.3.

Les scripts disposent également d'un certain nombre de méthodes destinées à simplifier la manipulation et le traitement des résultats obtenus. Ils sont présentés au tableau E.4.

E.2.2 Conservation des attributs

```
groupe.each { |tp|
  # Ecriture de la valeur
  tp[:identificateur|'identificateur'|variable] = resultat
  ...
  # Lecture de la valeur
  valeur_resultat = tp[:identificateur|'identificateur'|variable]
}
```

E.2.3 Utilisation d'un module

```
[res_module = ] nom_module( tp|groupe ) [{
  :nom_parametre_module >> valeur_parametre
```

Service	Description	Classe du résultat
`attributs`	Utilisé sur un objet de résultat, retourne la liste des attributs de cet objet.	`Array`
`groupe`	Le groupe à corriger.	`ContexteGroupe`
`parametres`	Les paramètres reçus depuis la ligne de commande. Si aucun paramètre n'était présent, ne contient aucune entrée.	`Hash`
`methodes`	Utilisé sur un objet de résultat, retourne la liste des méthodes de cet objet. Peut également être utilisé seul dans le script de correction pour obtenir la liste des méthodes offertes par l'environnement de correction Oto.	`Array`
`rep_script`	Le chemin absolu du répertoire temporaire contenant une copie du script de correction et des fichiers fournis par l'enseignant.	`String`
`rep_base`	Le chemin absolu du répertoire à partir duquel Oto a été invoqué.	`String`

Tableau E.2 Interface de programmation des scripts.

Méthode	**Description**
`commentaires`	Retourne un *hash* ordonné contenant les commentaires de résultats individuels dans le groupe.
`each`	Parcours les TP du groupe en exécutant, pour chacun, le bloc passé en paramètres. L'objet TP en cours de correction est passé en paramètre au bloc. Le répertoire contenant le TP est sélectionné comme répertoire courant. Le répertoire qui était courant avant l'exécution de la méthode le redevient une fois celle-ci complétée.
`effacer_resultats`	Supprime les résultats stockés dans chacun des TP du groupe. Cette fonctionnalité est utile pour supprimer les résultats déjà obtenus lorsqu'un test est exécuté successivement sur tous les TP d'un groupe et ensuite répété avec des paramètres ou des conditions différentes. Par exemple, la correction d'un TP de programmation parallèle sera possiblement répétée pour un nombre variable de processeurs. Dans un tel cas, il peut être nécessaire d'effacer les résultats déjà obtenus si l'identificateur du résultat sera appelé à changer, ce qui évitera d'inclure les résultats précédents dans le rapport de correction.
`groupe`	Donne un accès direct aux TP du groupe sous la forme d'un *hash* ordonné. La clef est le nom relatif du répertoire du TP et la valeur un objet de type `ContexteTP`.

Tableau E.3 Méthodes de l'objet `groupe`.

Méthode	Description
`ajouter_fichier( chemin_absolu_fichier, chaîne )`	Ajouter une chaîne de caractères à la fin d'un fichier. Si le fichier n'existe pas, il est créé.
`lire_fichier( chemin_absolu_fichier )`	Lire un fichier comme une chaîne de caractères.
`ecrire_fichier( chemin_absolu_fichier, chaîne )`	Écrire le contenu d'une chaîne dans un fichier texte. Si le fichier n'existe pas, il est créé, s'il existe déjà, son contenu est écrasé.

Tableau E.4 Services supplémentaires destinés au traitement des résultats dans les scripts.

```
  ...
}]
```

E.2.4 Rapports

Les rapports peuvent recevoir une ou plusieurs valeurs. Dans ce dernier cas, les valeurs seront combinées pour produire le rapport. Les résultats de groupe apparaîtront dans le rapport produit dans le même ordre qu'ils sont passés entre parenthèses dans les paramètres de l'appel du module-rapport.

```
[res_rapport = ] nom_rapport( groupe|resultat_de_groupe,... ) [{
  :nom_parametre_rapport >> valeur_parametre
  ...
}]
```

E.2.5 Commandes

Dans les scripts Oto, les commandes peuvent être utilisées tant à l'intérieur qu'à l'extérieur du `groupe.each`. Leurs résultats peuvent être conservés dans l'objet `tp`.

```
` commande parametres `
[res_commande = ] nom_commande_bash ['param1',...]
```

E.2.6 Contrôle de l'exécution

Les scripts Oto fournissent des méthodes permettant de contrôler l'exécution. Dans le DSL Oto, les seuls scripts dans lesquels l'échec d'une assertion doit arrêter leur exécution seront ceux utilisés à la création d'une boîte. Dans un tel cas, l'échec d'une assertion annule la remise d'un TP dans cette boîte.

Le module `Pass` facilite la manipulation des assertions. Le traitement effectué selon la condition varie en fonction de la méthode utilisée. `Pass.if` réussit si la condition est vraie. Au contraire, `Pass.unless` réussit si la condition est fausse.

Les paramètres des deux cas sont similaires. En premier lieu, nous trouvons la condition, c'est-à-dire l'expression à évaluer. Ensuite, il sera possible de fournir, optionnellement, le nom de l'exception qui sera soulevée, suivi possiblement de zéro à plusieurs paires de chaînes de caractères ou de symboles. Ces paires sont autant de détails sur l'exception qui seront affichés à l'utilisateur sur le canal d'erreur.

```
Pass.if( condition [,:NOM_EXCEPTION_SOULEVEE] [,'detail1', 'explication1',...] )
Pass.unless( condition [,:NOM_EXCEPTION_SOULEVEE] [,'detail1', 'explication1',...] )
```

E.2.7 Limitation de la liste de résultats inclus dans le script

Il sera parfois souhaitable de ne pas inclure dans le rapport toutes les informations retournées dans le résultat de l'exécution d'un module. Pour ce faire, il est possible d'utiliser la méthode `inclure_resultat` pour spécifier quels résultats parmi ceux retournés par le module dans l'objet résultat devant être inclus.

```
groupe.each { |tp|
  # Ecriture de la valeur
  res_module = nom_module( tp|groupe )
  res_module.inclure_resultat( :resultat1|'resultat1'[,...] )
}
```

E.2.8 Substitution de commentaires de résultats pour le rapport

```
groupe.each { |tp|
```

```
  # Ecriture de la valeur
  tp[:identificateur|'identificateur'] = nom_module( tp|groupe )
}
groupe.commentaire_resultat( :identificateur|'identificateur'
                             :texte_substitué|'texte_substitué')
```

E.3 Résultats

Les tableaux E.5 et E.6 contiennent respectivement les principales méthodes disponibles sur les objets de résultats individuels et collectifs.

E.4 Autres méthodes utiles

Nous présentons au tableau E.7 d'autres méthodes facilitant le développement de scripts Oto, particulièrement au niveau de la manipulation des fichiers et des répertoires.

Méthode	**Description**
`execute?`	Si l'exécution du module a été réussie, vrai, sinon faux.
`non_execute?`	Si l'exécution du module n'a pas été réussie, vrai, sinon faux.
`reussi?`	Si l'objectif du module a été atteint, vrai, sinon faux.
`echoue?`	Si l'objectif du module n'a pas été atteint, vrai, sinon faux.
`nom_module`	Retourne le nom du module.
`type_module`	Retourne le type du module : individuel, collectif ou commande *bash*.
`description_module`	Retourne la description du module.
`commenter_resultat(` *nom_resultat*, *commentaire* `)`	Permet de substituer au nom d'un résultat un commentaire.
`resultats`	Retourne un *hash* contenant les résultats.
`inclure_resultat(` *nom_resultat|liste_noms_resultats* `)`	Permet de spécifier un ou plusieurs résultats qui seront inclus dans les résultats formattés. Par défaut, tous les résultats sont inclus. Toutefois, après avoir utilisé cette méthode, seuls les résultats spécifiés en paramètres seront inclus.
`obtenir_commentaire(` *identificateur* `)`	Retourne le commentaire associé au résultat correspondant à l'identificateur, ou *nil* si celui-ci n'a pas été créé.
`resultats_formattes`	Retourne un *hash* contenant les résultats en tenant compte de la réussite ou de l'échec de l'exécution, des commentaires et des la liste d'inclusion (s'il y a lieu).

Tableau E.5 Méthodes de l'objet de résultat individuel.

Méthode	**Description**
`execute?`	Si l'exécution du module a été réussie, vrai, sinon faux.
`non_execute?`	Si l'exécution du module n'a pas été réussie, vrai, sinon faux.
`reussi?(` *tp* `)`	Si l'objectif du module a été atteint, vrai, sinon faux.
`echoue?(` *tp* `)`	Si l'objectif du module n'a pas été atteint, vrai, sinon faux.
`nom_module`	Retourne le nom du module.
`type_module`	Retourne le type du module : individuel, collectif ou commande *bash*.
`description_module`	Retourne la description du module.
`description_reussi`	Retourne ce qui doit être vrai pour qu'un TP soit classé dans la liste des « réussi ».
`description_echoue`	Retourne ce qui doit être avoir été échoué pour qu'un TP soit classé dans la liste des « échoué ».
`commenter_resultat(` *nom_resultat*, *commentaire* `)`	Permet de substituer au nom d'un résultat un commentaire.
`obtenir_resultat(` *nom_resultat*, tp `)`	Retourne le résultat pour le TP mentionné.
`obtenir_resultats(` *nom_resultat* `)`	Retourne un *hash* contenant le résultat pour tous les TP concernés. Les clés du *hash* sont le répertoire respectif de chaque TP.

Tableau E.6 Méthodes de l'objet de résultat collectif.

Méthode	**Description**
fichier.`abs`	Retourne le chemin absolu du fichier sur lequel cette méthode est utilisée.
fichier.`crFich`	Utilisé sur une chaîne de caractères, crée un fichier éponyme. La méthode soulève une exception si un fichier de ce nom existe déjà.
repertoire.`crRep`	Utilisé sur une chaîne de caractères, crée un répertoire éponyme. La méthode soulève une exception si un répertoire de ce nom existe déjà.
fichier.`detruire`	Supprime le fichier sur lequel cette méthode est utilisée.
fichier.`existe?`	Vérifie si le fichier sur lequel cette méthode est utilisée existe.
repertoire.`exporterVers(` *repExterne* `)`	Utilisée sur un répertoire, copie le contenu de celui-ci dans le répertoire passé en paramètre.
repertoire.`importer(` *externes* `[,...] )(` *repExterne* `)`	Utilisée sur un répertoire, copie dans celui-ci le contenu du ou des répertoires passés en paramètres.

Tableau E.7 Autres méthodes utiles au développement de scripts Oto.

BIBLIOGRAPHIE

(1) Aleksi Ahtiainen, Sami Surakka et Mikko Rahikainen : Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. *In Baltic Sea '06: Proceedings of the 6th Baltic Sea conference on Computing education research*, pages 141–142, 2006.

(2) Anthony Allowatt et Stephen Edwards : IDE support for test-driven development and automated grading in both Java and C++. *In eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 100–104, New York, NY, USA, 2005. ACM.

(3) Gregory R. Andrews : *Foundations of Multithreaded, Parallel, and Distributed Programmin.* Addison-Wesley Longman Publishing Co., Inc., 2000.

(4) Christian Arwin et S. M. M. Tahaghoghi : Plagiarism detection across programming languages. *In ACSC '06: Proceedings of the 29th Australasian Computer Science Conference*, pages 277–286, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

(5) Kent Beck et Erich Gamma : Test-infected: programmers love writing tests. pages 357–376, 2000.

(6) Katrin Becker : First principles of CS instruction. *J. Comput. Small Coll.*, 22(2):77–84, 2006.

(7) Brenda Cheang, Andy Kurnia, Andrew Lim et Wee-Chong Oon : On automated grading of programming assignments in an academic institution. 41(2):121–131, 2003.

(8) Charlie Daly et Jane Horgan : Patterns of plagiarism. *SIGCSE Bull.*, 37(1):383–387, 2005.

(9) Christopher Douce, David Livingstone et James Orwell : Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3):4, 2005.

(10) Stephen H. Edwards : Using software testing to move students from trial-and-error to reflection-in-action. *In SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 26–30, New York, NY, USA, 2004. ACM.

(11) George E. Forsythe et Niklaus Wirth : Automatic grading programs. *Commun. ACM*, 8(5):275–278, 1965.

(12) Sandra P. Foubister, Greg Michaelson et N. Tomes : Automatic assessment of elementary Standard ML programs using Ceilidh. *J. Comp. Assisted Learning*, 13(2):99–108, 1997.

(13) Martin Fowler : A pedagogical framework for domain-specific languages. *IEEE Software*, 26(4):13–14, 2009.

(14) Xiang Fu, Boris Peltsverger, Kai Qian, Lixin Tao et Jigang Liu : Apogee: automated project grading and instant feedback system for web based computing. *In SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 77–81, New York, NY, USA, 2008. ACM.

(15) David Gitchell et Nicholas Tran : Sim: a utility for detecting similarity in computer programs. *SIGCSE Bull.*, 31(1):266–270, 1999.

(16) Dick Grune et Matty Huntjens : Detecting copied submissions in computer science workshops. *Informatie* (en Hollandais), 31(11):864–867, novembre 1989.

(17) Frédéric Guérin : Oto, un outil générique et extensible pour corriger les travaux de programmation. Mémoire de maîtrise, Dép. d'Informatique, Université du Québec à Montréal, oct. 2005. `http://www.info2.uqam.ca/~tremblay/Maitrises/guerin.html`.

(18) J. B. Hext et J. W. Winings : An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5):272–275, 1969.

(19) Colin A. Higgins, Geoffrey Gray, Pavlos Symeonidis et Athanasios Tsintsifas : Automated assessment and experiences of teaching programming. *J. Educ. Resour. Comput.*, 5(3):5, 2005.

(20) M. Joy, P.-S. Chan et M. Luck : Networked submission and assessment. *In Proceedings of the 8th Annual Conference on the Teaching of Computing*, pages 41–45. LTSN Center for Information and Computer Sciences, 2000.

(21) M. Joy et M. Luck : Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(1):129–133, 1999.

(22) E. Labonté : Outil de correction semi-automatique de programmes Java. Mémoire de maîtrise, Dép. d'Informatique, Université du Québec à Montréal, décembre 2002. `http://www.info2.uqam.ca/~tremblay/Maitrises/labonte.html`.

(23) Marjan Mernik, Jan Heering et Anthony M. Sloane : When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

(24) George A. Miller : The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.

(25) Jakob Nielsen et Rolf Molich : Heuristic evaluation of user interfaces. *In CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256, New York, NY, USA, 1990. ACM.

(26) L. Prechelt, G. Malpohl et M. Philippsen : Finding plagiarisms among a set of programs with JPlag. Non publié. Soumis au Journal of Universal Computer Science, 2000.

(27) Saul Schleimer, Daniel S. Wilkerson et Alex Aiken : Winnowing: local algorithms for document fingerprinting. *In SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.

(28) Diomidis Spinellis : Reliable software implementation using domain specific languages. *In* G. I. Schuëller et P. Kafka, éditeurs : *Proceedings ESREL '99 — The Tenth European Conference on Safety and Reliability*, pages 627–631, Rotterdam, septembre 1999. ESRA, VDI, TUM, A. A. Balkema.

(29) Diomidis Spinellis : Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, février 2001.

(30) Mark Strembeck et Uwe Zdun : An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, août 2009.

(31) M. Takim : Applications *Web* pour l'utilisation des services de l'outil Oto. Rapport de projet, Dép. d'Informatique, Université du Québec à Montréal, décembre 2006. `http://www.info2.uqam.ca/~tremblay/Maitrises/takim.html`.

(32) David Thomas et Andrew Hunt : *Programming Ruby: the pragmatic programmer's guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

(33) G. Tremblay, F. Guérin, A. Pons et A. Salah : Oto, a generic and extensible tool for marking programming assignments. *Software—Practice and Experience*, 38(3):307–333, March 2008.

(34) G. Tremblay, L. Laforest et A. Salah : Extending a marking tool with simple support for testing (*Poster*). *In Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education*, page 313. ACM, June 2007.

(35) G. Tremblay, M. Takim et A. Salah : Oto, un outil d'aide à la correction de programmes : Guide d'utilisation destiné aux étudiants. septembre-décembre 2006.

(36) Guy Tremblay : La correction des travaux de programmation et la détection des cas de plagiat: Qu'est-ce qui se fait ailleurs? mai 1999.

(37) Kristina L. Verco et Michael J. Wise : Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. *In* John Rosenberg, éditeur : *Proceedings of the First Australian Conference on Computer Science Education*, pages 81 – 88, Sydney, Australia, July 3-5 1996. SIGCSE, ACM.

(38) Michael J. Wise : YAP3: improved detection of similarities in computer program and other texts. *SIGCSE Bull.*, 28(1):130–134, 1996.